Commodore
Sachbuch

Horst-Rainer Henning

Grafik mit AMIGA-BASIC

Programmierung von Grafik unter Verwendung der Amiga-Systemroutinen: Icon-Programmierung ★ Transformationen ★ Spezial-Modi ★ Grafikspeicherung ★ Apfelmännchen ★ 3-D-Grafik ★ Animation ★ HAM

Auf der beiliegenden Diskette
finden Sie über 50 Beispielprogramme.



Grafik mit AMIGA-BASIC

Grafik mit AMIGA-BASIC

Programmierung von Grafik unter Verwendung der Amiga-Systemroutinen: Icon-Programmierung ★ Transformationen ★ Spezial-Modi ★ Grafikspeicherung ★ Apfelmännchen ★ 3-D-Grafik ★ Animation ★ HAM

Horst-Rainer Henning

Commodore Sachbuch



Markt&Technik Verlag AG

CIP-Titelaufnahme der Deutschen Bibliothek

Henning, Horst-Rainer:
Grafik mit AMIGA-BASIC : Programmierung von Grafik unter Verwendung der Amiga-Systemroutinen:
Icon-Programmierung – Transformationen – Spezial-Modi – Grafikspeicherung – Apfelmännchen – 3-D-Grafik –
Animation – HAM / Horst-Rainer Henning. –
Haar bei München : Markt-u.-Technik-Verl., 1989.
(Commodore-Sachbuch)
ISBN 3-89090-669-9

Dieses Buch wurde mit Microsoft Word 4.0 bearbeitet und über Compugrafik belichtet.

4 5 6 7 8 9 10 11 12 13 14 15
90 91 92

ISBN 3-89090-669-9

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Wiener Verlag, Himberg
Printed in Austria

Inhaltsverzeichnis

2. Teil Darstellung

3. Teil View-Modi

4. Teil Spezial

5. Teil Anhang

Vorwort

Wenn Sie Ihren Amiga beschreiben sollen, werden Sie sicherlich sehr schnell von seinen phantastischen Grafikfähigkeiten zu schwärmen anfangen. Die wirklich interessanten Grafikeigenschaften jedoch kennen Sie wahrscheinlich nur von Demonstrations-Programmen oder von professioneller Software her. Doch wenn Sie als eifriger Basic-Programmierer diese tollen Eigenschaften nutzen wollen, stoßen Sie auf Schwierigkeiten. In den meisten Büchern zu diesem Thema bekommen Sie nur C-Programme serviert. Und wenn Sie dann einmal ein Basic-Programm mit speziellen Grafik-Effekten in die Hand bekommen, läuft es so langsam, daß Sie schnell die Lust daran verlieren können. Soll das Tor zu der wunderbaren Grafik-Welt des Amiga dem Basic-Programmierer verschlossen bleiben? Denkt denn niemand an die »Minderheit« von 90% der Amiga-Besitzer, die hauptsächlich in Basic programmieren?

Nun, auch wenn es etwas marktschreierisch klingt, dieses Buch wurde für diese »Minderheit« geschrieben. Das Basic des Amiga bietet an sich durch eine Vielzahl von Grafik-Befehlen schon allerhand zu diesem Thema. Man muß nur wissen, wie man sie nutzen, sprich programmieren kann. Das Buch will dazu Hilfestellung geben und Problemlösungen aufzeigen. Mit diesen Grundlagen beschäftigt sich der erste Teil des Buches.

Außerdem finden Sie in diesem Buch grafische Besonderheiten, die das normale Amiga-Basic nicht bietet. Den Schlüssel dazu finden Sie in den Routinen des Betriebssystems. Diese Library-Routinen können von den verschiedenen Programmiersprachen wie C oder Assembler aufgerufen werden. Auch das Programm Amiga-Basic nutzt weidlich diese Routinen. Leider werden dabei eine Vielzahl interessanter Routinen nicht berücksichtigt. Zum Glück wurde jedoch eine Möglichkeit offengehalten, mit der man vom Basic heraus auf diese Bibliotheks-Routinen zugreifen kann.

Ab dem zweiten Teil dieses Buches wird mit diesen Library-Routinen gearbeitet. Ab jetzt stehen Ihnen als Leser also fast alle Möglichkeiten offen. Dabei ist die Funktion der Routinen nur ein Teilaspekt. Ebenso wichtig ist es, daß damit die Techniken der Grafik-Programmierung in einer neuen Dimension gezeigt werden können. Beim

Zeichnen sind Sie nicht mehr auf ein Window angewiesen. Mühelos zeichnen Sie Ihre Grafiken mit Text direkt in den Screen. Im Bruchteil einer Sekunde setzen Sie mit maximal 32 Datenwerten eine neue Farbtabelle. Sie sehen, die Farbgebung muß nicht statisch sein. Im Gegenteil, Sie können alleine durch die Farbänderung eine Animation der Grafik erreichen.

Aber damit fängt es eigentlich erst an. Liniengrafiken werden verschoben, verkleinert, vergrößert, gespiegelt und gedreht. Dabei erfahren Sie auch, was es mit dem Double Buffering auf sich hat, und natürlich, wie das alles funktioniert. Dabei bringt die Liniengrafik nur einen Vorgeschmack auf das Folgende. Die Lupentechnik vergrößert oder verkleinert komplette Bildschirmbereiche. Neben der Spiegelung von vollständigen oder Teil-Grafiken bietet die Verzerrung derselben einige reizvolle Gesichtspunkte. Obwohl bei dieser Art der Transformationen jedes Pixel einen neuen Platz erhält, geschieht das in Sekundenbruchteilen. Die Grundlagen der dreidimensionalen Grafik dürfen natürlich auch nicht fehlen.

Auch die speziellen Modi des Amiga sind ausreichend berücksichtigt. So können Sie Riesengrafiken erstellen, die größer sind als der sichtbare Bildschirm. Damit haben Sie die Grundlage für ein echtes Playfield scrolling. Ebenso interessant ist die Programmierung von zwei unabhängigen Bildebenen im Modus Dual Playfield. Wissen Sie, daß der Amiga 64 Farben gleichzeitig ausgeben kann? Der ExtraHalfBrite-Modus bietet Ihnen dazu die Möglichkeit. Gleich 4096 verschiedene Farben können Sie im HoldAndModify-Modus auf dem Bildschirm bringen. Das ist natürlich längst nicht alles, was der Amiga an Grafikfähigkeiten zu bieten hat. Lassen Sie sich überraschen, denn mehr soll an dieser Stelle nicht verraten werden.

Doch was nützen Ihnen die ganzen Grafik-Möglichkeiten, wenn Sie das Ergebnis der Programmierung nicht festhalten können? Auch dafür ist Vorsorge getroffen. Sie entwickeln in einem extra Kapitel ein Programm, mit dem Sie schnell und bequem fast alle Grafiken einlesen und speichern können. Das Programm speichert nicht nur komplette Bildschirme, sondern auch Ausschnitte daraus und scheitert auch nicht an den speziellen Modi SUPERBITMAP, HAM oder EXTRAHALFBRITE. Außerdem erfahren Sie noch, was es mit den Icons auf sich hat und wie Sie die Piktogramme mit zwei verschiedenen Bildern konstruieren können. Die fraktale Grafik wurde auch nicht vergessen. Wie Sie mit Hilfe des Coppers tausend Farben und mehr mit nur einer einzigen Bit-Ebene auf den Bildschirm zaubern können, erfahren Sie in einem weiteren Kapitel.

All das, was Sie bisher gelesen haben, und noch mehr finden Sie in diesem Buch. Die einzelnen Techniken sind ausreichend und verständlich erklärt. Dabei bleibt die Programmierung der Techniken immer im Vordergrund. Denn was nützt Ihnen die schönste Theorie, wenn Sie sie nicht in die Praxis umsetzen können? Die in dem Buch besprochenen Basic-Befehle, System-Routinen und Techniken finden Sie daher alle in

kleineren oder größeren Beispielen und Programmen verarbeitet. Insgesamt sind es über 80 Programme, die Ihnen die praktische Seite nahe bringen werden. Nur so erhalten Sie die Sicherheit, selbst lauffähige Programme schreiben zu können. Damit Sie sich nicht der Mühe des Abtippens unterziehen müssen, finden Sie alle Programme auf der beiliegenden Diskette, deren Handhabung im Anhang III genau erklärt wird.

Zwei Gesichtspunkte machen meiner Meinung nach, neben der Farbgebung und Auflösung der Bilder, die Grafik erst richtig interessant. Ein Punkt davon ist die Geschwindigkeit, mit der die Grafiken erstellt oder manipuliert werden. Der zweite Gesichtspunkt ist die Bewegung von Grafik in jeder Form. Da ich gerne Aktion auf dem Bildschirm habe, halte ich nicht viel von Grafiken, die sich Pünktchen für Pünktchen aufbauen und dann starr auf dem Bildschirm herumstehen. Wenn irgendwie möglich, habe ich Tempo in die Sache gebracht. Natürlich, ein sauberer Programmaufbau ist wichtig. Darunter soll aber nicht das Ergebnis leiden. Wenn Sie ein Programm vorführen, das Ihren Zuschauern ein Gähnen nach dem anderen entlockt, wird sich bald niemand mehr für Ihre Programme interessieren. Daher wird bei den Programmen das Wichtigste, nämlich das Ergebnis, nie aus dem Auge verloren.

Haben Sie inzwischen Lust bekommen, die Grafik-Programmierung auf dem Amiga zu versuchen? Prima! Dann überlegen wir noch schnell, was Sie sonst noch brauchen. Natürlich einen Amiga 500, 1000 oder 2000 mit 512 Kbyte Speicher und natürlich das Amiga-Basic von Ihrer Basic-Diskette. Ein zweites Diskettenlaufwerk wäre hilfreich, ist aber nicht unbedingt erforderlich. Ein Programm benötigt einen Joy-Stick.

Haben Sie das alles? Nun, auf was warten wir dann noch? Packen wir's an.

Vorher bedanke ich mich recht herzlich bei allen, die mich bei der Erstellung des Buches unterstützt haben. Mein besonderer Dank gilt meiner Frau Lisbeth, Frau Christine Baumann und Herrn Hergenröder.

Mit tiefer Trauer vernahm der Verlag den völlig unerwarteten Tod von Horst-Rainer Henning. Er verstarb im Alter von 45 Jahren am 29. Dezember 1988.

Horst-Rainer Henning zählte zu den Amiga-Pionieren in Deutschland und war einer der besten Kenner von Amiga-Basic. Seiner bis zuletzt ungebrochenen Begeisterung für diesen Computer hat der Verlag drei fundierte Bücher zu verdanken, die alle eine breite Akzeptanz am Markt gefunden haben und weiterhin finden werden.

Der Verlag sieht sich im Sinne des Autors verpflichtet, auch in Zukunft seine Bücher einem breiten Publikum anzubieten und unvollendete Werke zu einem Abschluß zu bringen.

Lektorat
Verlagsleitung — Januar 1989

Grafik mit AMIGA-BASIC

Grundlagen

1. Teil

Commodore Sachbuch

Kapitel 1

Aller Anfang ist nicht schwer

Im ersten Kapitel finden Sie, wie Sie sicherlich erwartet haben, die wichtigsten Grundlagen der Grafik-Programmierung. Es werden die einfachen Grafik-Befehle vom Punkt bis zum Kreis vorgestellt. Wenn für Sie die Begriffe Bildschirm-Koordinaten, Pixel, Linie, Rechteck, Kreis und Ellipse ein alter Hut sind, können Sie das Kapitel überspringen. Fühlen Sie sich dagegen nicht hundertprozentig sicher, so lesen Sie es lieber durch.

1.1 Bildschirm-Koordinaten

Wenn Sie auf einem Blatt Papier einen Punkt setzen, so ist dieser eine bestimmte Strecke von der unteren Papierkante und eine bestimmte Entfernung von der linken Papierkante entfernt. In der Mathematik sagt man, diese beiden Strecken sind Koordinaten eines Punktes. Damit haben Sie bereits ein Koordinatensystem festgelegt. Der Nullpunkt des Koordinatenkreuzes befindet sich bei diesem Beispiel in der linken unteren Ecke. Die Strecke vom Nullpunkt nach rechts ist die X-Koordinate und vom Nullpunkt nach oben ist die Y-Koordinate des Punktes. Dieses Koordinatensystem nennt man kartesisches Koordinatensystem (nach R. Descartes, 1596-1650). Es ist das am häufigsten verwendete Koordinatensystem.

Auch bei der Computergrafik kommen Sie nicht ohne Koordinaten aus. Dabei gibt es zwei gravierende Unterschiede zu den Koordinaten der allgemeinen Mathematik. Der Nullpunkt befindet sich nämlich nicht in der linken unteren Ecke, sondern in der linken oberen Ecke des Bildschirmes. Diese linke obere Ecke wird Ihnen bei der Grafik-Programmierung häufig begegnen. Sie ist nicht nur für die Koordinaten maßgebend, sondern auch das Speichern eines Bildes fängt an diesem Punkt an. Die Daten eines Sprites beginnen an der linken oberen Ecke, ebenso der Bildausschnitt für eine GET-Anweisung. Wenn irgendein rechteckiger Bereich benötigt wird, beginnt, auch bei den Routinen des Betriebssystems, die Festlegung mit der linken oberen Ecke.

Die Entfernung der Koordinaten wird bei einem Computer in Bildpunkten, auch Pixel genannt, angegeben. Dabei erhält der erste Bildpunkt, und das ist der zweite Unterschied zur Mathematik, den Wert Null. In Bild 1.1 ist das Prinzip an den beiden

Koordinatenpaaren einer Linie dargestellt. Die Streckenangabe in Pixeln ist allerdings nicht ganz so unproblematisch, wie sie auf den ersten Blick scheint. So ein Grafikpunkt ist leider nicht quadratisch. Je nach dem gewählten Modus ergeben sich dabei verschiedene Verhältnisse der X- zu den Y-Werten. Wir werden dieses Problem bei den Grafik-Modi lösen. Beim Start eines Basic-Programmes befinden Sie sich im Hires-Modus, dem Modus mit der höchsten Auflösung. Als Auflösung bezeichnet man die Anzahl der maximal darstellbaren Grafikpunkte. Im Hires-Modus ist Ihr Bildschirm 640 Bildpunkte breit und 256 (200) Pixel hoch. Das Basic-Ausgabefenster ist allerdings nur 200 Pixel hoch. Eine komplette Reihe von Bildpunkten bezeichnet man auch als Grafik-Zeile.

1.2 Ein kleiner Punkt

Nachdem Sie nun das Koordinaten-System kennengelernt haben, können Sie auch gleich Ihren ersten Grafik-Befehl ausprobieren. Die Anweisung

```
PSET<STEP> (x,y)<,Farbe>
```

zeichnet bei den Koordinaten *x* und *y* einen Punkt. Wenn Sie für die Option *Farbe* keinen Wert angeben, wird er mit der Vordergrundfarbe weiß gezeichnet. Wenn Sie die Farben nicht mit den Preferences auf der Workbench geändert haben, stehen Ihnen folgende Farben zur Verfügung:

Farbnummer 0 = blau (Hintergrundfarbe)
Farbnummer 1 = weiß (Vordergrundfarbe)
Farbnummer 2 = schwarz
Farbnummer 3 = orange

Diese kurze Zusammenstellung soll uns vorläufig genügen. Wir werden später detailliert auf die Farbgebung des Amiga eingehen. Kommen wir wieder auf die PSET-Anweisung zurück. Die Koordinaten für den zu setzenden Punkt müssen ganze (integer = %) Zahlen sein. Es können schließlich keine halben oder viertel Punkte gezeichnet werden. Haben Sie in Ihrem Basic-Programm nichts anderes programmiert, werden jedoch alle Zahlen als Zahlen mit einfacher Genauigkeit (!) definiert. Sie brauchen aber keine Angst zu bekommen, daß Sie beim Setzen eines Punktes etwas falsch machen können. Der Basic-Interpreter schneidet einfach die Nachkommawerte ab. Für die Programmierung mit den Routinen des Betriebssystems sollten Sie sich die richtige Variablen-Definition merken. Nun aber genug der Theorie. Setzen Sie nun Ihre ersten Grafikpunkte:

```
FOR i=100 TO 200:PSET (i,90):NEXT
```

Mit dieser kurzen Programmzeile wird eine Reihe von 101 Grafikpunkten in der Vordergrundfarbe gesetzt. Wollen Sie die Linie wieder verschwinden lassen, ohne mit der

CLS-Anweisung gleich den ganzen Bildschirm zu löschen, dann setzen Sie einfach die gleichen Positionen mit der Hintergrundfarbe:

```
FOR i=100 TO 200:PSET (i,90),0:NEXT
```

Falls Sie einmal die Hintergrundfarbe nicht mehr wissen und das kann bei einer größeren Grafik durchaus passieren, hilft Ihnen eine weitere Basic-Anweisung:

```
PRESET<STEP> (x,y)<,Farbe>
```

Wird PRESET ohne Farbe aufgerufen, so wird für Farbe die Hintergrundfarbe eingesetzt. Ansonsten gilt das für die PSET-Anweisung Gesagte auch für PRESET.

```
FOR i=50 TO 300:PSET (i,i/2):NEXT
FOR i=50 TO 300:PRESET (i,i/2):NEXT
```

Die beiden Zeilen zeichnen eine schräge Linie aus einzelnen Pixeln in der Vordergrundfarbe und löschen diese anschließend wieder. In den kleinen Beispielen haben wir die PSET-Anweisung zum Zeichnen von Linien mißbraucht. In der Praxis werden Sie die Anweisungen dafür nicht einsetzen. Die Arbeit nimmt Ihnen die Hardware des Amiga ab. Zu der schnellen und einfachen LINE-Anweisung kommen wir anschließend. Für Freihandzeichnungen in einem Malprogramm, oder zur grafischen Darstellung von Kurven kommen Sie jedoch ohne die PSET-Anweisung nicht aus. Das erste Programm-Beispiel zeichnet mit der PSET-Anweisung die drei gebräuchlichsten Winkelfunktionen:

```
REM Winkelfunktionen  Pfad: Grundlagen/1AllerAnfang/WiFu
'P1-1 **Beispiel für PSET**
pi=3.141592
v=80      'Vergroesserung
xk=50     'X-Koordinate
yk=90     'Y-Koordinate
FOR i = 0 TO xk + v*2*pi
  PSET (i,yk)
NEXT
PSET (xk,0)
FOR i = 0 TO 180
  PSET STEP (0,1)
NEXT
'Winkelfunktion y=sinx
FOR x= 0 TO 2*pi STEP .01
  PSET (xk+v*x,yk-(v*SIN(x)))
NEXT
'Winkelfunktion y=cosx
FOR x= 0 TO 2*pi STEP .01
```

```
  PSET (xk+v*x,yk-(v*COS(x))),3
NEXT
'Winkelfunktion y=tgx
fa=3:far=1
FOR x= Ø TO 2*pi STEP .Ø1
  yw=yk-(v*TAN(x))
  IF yw >=Ø THEN PSET (xk+v*x,yk-(v*TAN(x))),fa
  SWAP fa,far
NEXT
```

Zuerst wird das Koordinaten-Feld durch zwei, sich im Nullpunkt (für die Kurven) schneidende Geraden, festgelegt. Die erste, die X-Achse wird dabei durch eine Reihe von Punkten gezeichnet, wie Sie es in den Beispielen gesehen haben. Bei der Y-Achse wird jedoch die Option STEP eingesetzt. Diese Option bedeutet, daß die aktuelle Position zur Ausgangsposition für die folgende Anweisung wird. Im Programm wird mit

```
PSET (xk,Ø)
```

ein Pixel mit der X-Koordinate xk = 50 und mit der Y-Koordinate 0 gesetzt. Die folgende Anweisung

```
PSET STEP (Ø,1)
```

behält die X-Koordinate bei (Wert=0) und setzt eine neue Y-Koordinate, die +1 Pixel von der vorhergehenden Position entfernt ist. Durch die FOR/NEXT-Schleife wiederholt sich dieser Vorgang 181 Mal. Es folgt die Winkelfunktion y=sinx. Da die volle Sinus-Kurve gezeichnet werden soll, wird der maximale Wert der FOR/NEXT-Schleife auf 2*pi festgelegt. Durch STEP .01 wird die Kurve bildschirmgerecht auseinandergezogen. In der PSET-Anweisung wird der X-Wert mit dem Vergrößerungsfaktor *v* der Abstufung angepaßt. Der Y-Wert wird mit der SIN-Funktion errechnet und ebenfalls vergrößert.

Die nächste Winkelfunktion zeichnet die Cosinus-Kurve mit roten Punkten. Der Programmaufbau entspricht der vorhergegangenen Funktion. Die letzte Funktion y=tgx würde die Grenzen des Bildschirmes überschreiten und mit einer Fehlermeldung das Programm abbrechen. Das Programm prüft daher vor der Ausgabe des Punktes in der Variablen *yw*, ob die Grenzen des Bildes verlassen werden. In diesem Fall wird kein Punkt gesetzt. Durch die SWAP-Anweisung wird immer ein weißer und ein roter Punkt ausgegeben. Sie haben an diesem kurzen Programm gesehen, wie leicht es Ihnen der Amiga macht, einfache und auch komplexere Funktionen grafisch darzustellen.

1.3 Von der Linie zum Rechteck

Nun kommen wir zu einer Anweisung, die mit einer schier unglaublichen Geschwindigkeit arbeitet. 1 Millionen Bildpunkte werden in einer Sekunde gesetzt! Der 68000-Prozessor des Amiga alleine könnte diese atemberaubende Geschwindigkeit nicht schaffen. Er bedient sich eines Koprozessors, genannt Blitter, der die Arbeit selbständig übernimmt. Genug der Schwärmerei. Sehen Sie sich die LINE-Anweisung erst einmal in Ruhe an:

```
LINE <STEP> (x1,y1)-<STEP> (x2,y2)<,Farbe><,B<F>>
```

Durch die vielen Optionen sieht diese Anweisung auf den ersten Blick etwas gefährlich aus. Sie ist es aber nicht. Wenn Sie eine Linie ziehen wollen, müssen Sie nur den Anfangs- und Endpunkt der Linie als Koordinaten-Paare eingeben:

```
LINE (4,2)-(30,28)
```

Diese Befehlszeile zieht eine Linie von dem Bildpunkt mit den Koordinaten x=4 und y=2 zu dem Pixel x=30 und y=28. Die zeichnerische Darstellung dieser Linie finden Sie im Bild 1.1:

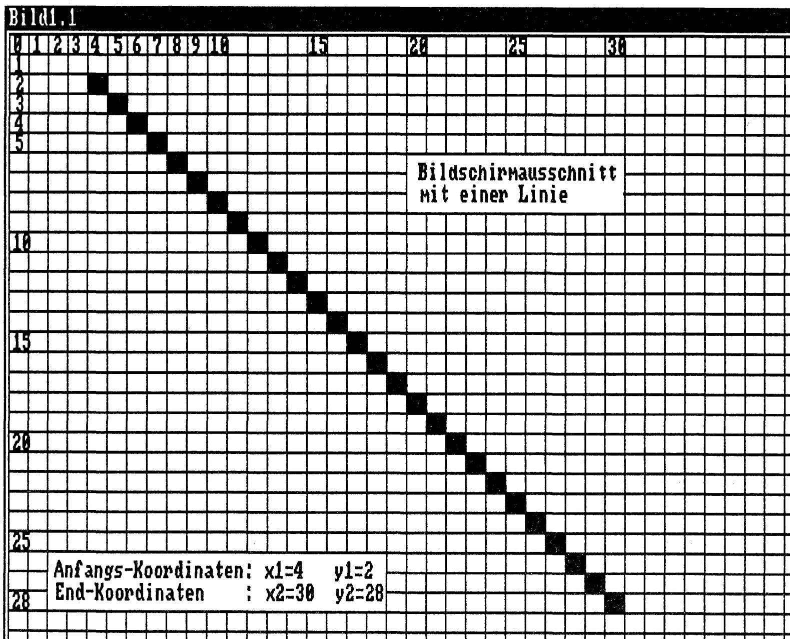


Bild 1.1: Bildschirmausschnitt mit einer Linie

Ohne Farbangabe, wie in dem Beispiel, wird die Linie in der Vordergrundfarbe weiß gezogen. Bei anderen Farbwünschen müssen Sie die Option *Farbe* versorgen. Die Farbangaben in diesem Kapitel gelten natürlich nur, wenn Sie die Farben nicht verändert haben. Wenn Sie mehrere Linien miteinander verbinden wollen, können Sie die gekürzte Form der Anweisung einsetzen:

```
LINE (100,20)-(400,20):LINE -(300,100):LINE -(100,20)
```

Mit diesen drei Anweisungen ziehen Sie ein Dreieck zwischen den Koordinaten 100/20, 400/20 und 300/100. Bei der Option STEP wird wieder die aktuelle Koordinate zum Ausgangspunkt für die nächste Anweisung, wie Sie es bei der Anweisung PSET gesehen haben. Schauen Sie sich nun das erste Beispiel zu der Anweisung LINE an:

```
REM Sonne  Pfad: Grundlagen/1AllerAnfang/Sonne
'P1-2 **Beispiel für LINE**
DEFINT a-z
d=90
DIM x1(d),x2(d),y1(d),y2(d)
pi!=3.141592
FOR i=0 TO 360 STEP 4
  w!=i*pi!/180
  r1=20:r2=280
  x1(i/4)=300+(r1*COS(w!)):y1(i/4)=90+(r1*SIN(w!)/2)
  x2(i/4)=300+(r2*COS(w!)):y2(i/4)=90+(r2*SIN(w!)/2)
NEXT
a=-1:fa=1:fa1=3:fa2=2
WHILE a
  FOR i=0 TO 90
    LINE (x1(i),y1(i))-(x2(i),y2(i)),fa
  NEXT i
  SWAP fa,fa1:SWAP fa,fa2
  ta-=INKEY-:IF ta-<>""THEN a=0
WEND
```

Damit bei diesem Programm das Zeichnen der Linien nicht durch zusätzliche Rechenzeit gebremst wird, werden die Anfangs- und Endpunkte der Linien vor dem eigentlichen Programmablauf berechnet und in Feldvariablen gespeichert. Dazu werden zuerst die Variablen der beiden jeweiligen Koordinaten-Paare dimensioniert. Nun werden die Koordinaten mit Hilfe der SIN- und COS-Funktion errechnet. Die Linien werden zueinander im Abstand von 4 Grad berechnet. Nachdem die Vorarbeiten beendet sind, braucht in der WHILE/WEND-Schleife nur noch die LINE-Anweisung zum Zeichnen

der Linien aufgerufen werden. Für die dreifachen Farbänderungen sorgen die beiden SWAP-Anweisungen. Das Programm kann durch Betätigung einer beliebigen Taste abgebrochen werden.

Das zweite LINE-Beispiel zeichnet bunte Bündel von Linien unterschiedlicher Länge, die kreuz und quer über den Bildschirm flitzen. In ähnlicher Form haben Sie das sicherlich schon einmal in einem Computer-Spiel bewundert.

```
REM Mikado  Pfad: Grundlagen/1AllerAnfang/Mikado
'P1-3
DEFINT a-z
z=15: DIM x(z),y(z),x1(z),y1(z)
xmax=WINDOW(2)-2:ymax=WINDOW(3)-2
PALETTE Ø,Ø,Ø,Ø:PALETTE 2,Ø,1,Ø
a=-1
WHILE a
  FOR i=Ø TO z
    LINE (x(i),y(i))-(x1(i),y1(i)),Ø
    x(i)=ABS(x(ia)+RND*3ØØ-15Ø)
    IF x(i)>xmax THEN x(i)=xmax
    y(i)=ABS(y(ia)+RND*6Ø-3Ø)
    IF y(i)>ymax THEN y(i)=ymax
    x1(i)=ABS(x1(ia)+RND*3ØØ-15Ø)
    IF x1(i)>xmax THEN x1(i)=xmax
    y1(i)=ABS(y1(ia)+RND*6Ø-3Ø)
    IF y1(i)>ymax THEN y1(i)=ymax
    LINE (x(i),y(i))-(x1(i),y1(i)),fa
    ia=i :fa=RND*2+1
  NEXT i
  IF INKEY-<>"" THEN a=Ø
WEND
```

Die Koordinaten der Anfangs- und Endpunkte der Linien werden durch Zufallszahlen ermittelt. Ergeben die Zufallszahlen, daß die Fenstergrenzen überschritten würden, so werden die Linien nur bis zu den Fenstergrenzen gezeichnet. Zu den WINDOW-Funktionen, mit denen die Grenzen ermittelt werden, kommen wir im nächsten Kapitel. Da die Koordinaten der Linien in Variablen-Feldern gespeichert sind, wird immer eine Linie mit der Hintergrundfarbe gelöscht und eine neue Linie mit der zufällig ermittelten Farbe *fa* gezeichnet. So sind immer nur 16 Linien gleichzeitig zu sehen.

Wenn Sie ein Rechteck zeichnen wollen, benötigen Sie zwei diagonal gegenüberliegende Punkte, oder einen beliebigen Punkt für die Position und zusätzlich die Höhe und die Breite des Rechteckes. Da die erste Voraussetzung in der LINE-Anweisung gegeben ist,

erhielt das Amiga-Basic keinen eigenen Block-Befehl. Es wird einfach die LINE-Anweisung mit einem b (für block) dahinter versehen und schon ist aus der Linien-Anweisung LINE eine Anweisung für ein Rechteck geworden:

```
LINE (100,30)-(500,150),,b
```

Diese Programmzeile zeichnet ein Rechteck mit den Koordinaten 100/30 für die obere linke Ecke und 500/150 für die untere rechte Ecke. Wollen Sie das Rechteck ausgemalt haben, so geben Sie zusätzlich die Option *f* für fill an:

```
LINE (40,10)-(100,20),3,bf
```

Probieren wir das Zeichnen von Rechtecken gleich an einem kleinen Geschicklichkeitsspiel aus:

```
REM Irrgarten  Pfad: Grundlagen/1AllerAnfang/Irrgarten
'P1-4 **Beispiel für LINE,bf**
FOR i=0 TO 170 STEP 20
  LINE (10,i)-(590,i+5),1,bf
NEXT
FOR i=10 TO 170 STEP 20
  LINE (0,i)-(580,i+5),1,bf
NEXT
LINE (591,0)-(620,170),1,bf
PRINT "START>    "
LOCATE 22,67:PRINT "    ZIEL"
warte: muss=MOUSE(0):x=MOUSE(1):y=MOUSE(2)
IF x<80 THEN
  IF y<10 AND x>50 THEN s=1
END IF
IF NOT s=1 THEN warte
start:
a=-1
WHILE a
  muss=MOUSE(0)
  IF POINT(MOUSE(1),MOUSE(2)) THEN
    BEEP:f=f+1
    LOCATE 23,1
    PRINT "Fehler: "f;
  END IF
  IF MOUSE(1)>530 AND MOUSE(2)>165 THEN a=0
WEND
LOCATE 1,1
```

Zuerst werden die Rechtecke für den Irrgarten gezeichnet. Das dürfte Ihnen inzwischen geläufig sein. Anschließend werden Start und Ziel gekennzeichnet. Bei dem Label »warte« wird die Position des Mauszeigers abgefragt. Befindet sich der Pointer in dem freien Rechteck neben dem Text START, so wird die Schleife verlassen. Nun beginnt bei dem Label »start« in der WHILE/WEND-Schleife der eigentliche Programmablauf. In dieser Schleife wird wieder die Position des Mauszeigers abgefragt. Die beiden Koordinaten der aktuellen Position setzen wir dabei in einer neuen Funktion ein:

```
v=POINT(x,y)
```

Diese Funktion liefert die Farbnummer eines Pixels mit den Koordinaten *x* und *y*. In dem Programm wird kein Wert auf eine bestimmte Farbe gelegt. Es wird nur geprüft, ob die Funktion einen logisch wahren Wert liefert, also die Farbnummer größer 0 (Hintergrundfarbe) ist. In diesem Falle wird mit einem BEEP ein Fehler angezeigt, die Fehler-Variable *f* um +1 erhöht und die Gesamtzahl der Fehler angezeigt. Zum Schluß wird in der Schleife geprüft, ob das Ziel erreicht ist. In diesem Fall wird die Variable *a* unwahr und die Schleife verlassen.

Das Spiel ist recht einfach. Sie bringen den Mauspfeil in das freie Feld rechts neben das Wort START. Damit beginnt das Spiel. Die Aufgabe des Spielers ist es, den Mauspfeil immer auf dem blauen Untergrund durch das Labyrinth zu bewegen. Sobald der Pfeil an die weiße Begrenzung des Labyrinths stößt, wird mit einem BEEP der Fehler registriert. Hat der Mauspfeil das Feld neben dem Wort ZIEL erreicht, ist das Spiel beendet. Die Anzahl der Fehlerpunkte sagt aus, wer der beste Spieler ist.

1.4 Vom Kreis zur Ellipse

Mindestens genauso vielseitig wie die LINE-Anweisung ist der Befehl zum Zeichnen von Kreisen und Ellipsen.

```
CIRCLE <STEP> (x,y),r<,Farbe<,Start,Ende<,Bild>>>
```

In dem folgenden Programm werden alle Möglichkeiten dieser Anweisung demonstriert. Bei der Programmbeschreibung am Ende des kleinen Programmes werden wir die einzelnen Optionen genauer unter die Lupe nehmen.

```
REM Circle  Pfad: Grundlagen/1AllerAnfang/Circle
'P1-5 **Beispiel für CIRCLE**
DEFINT a-z
x1=WINDOW(2)/2
y1=WINDOW(3)/2
pi!=3.141593
start:
CLS
```

```
PRINT  "Kreis=K  Ellipse=E  beenden=B "
ke–=UCASE–(INPUT–(1))
IF ke–="B" THEN ende
IF ke–="K" THEN Kreis
IF ke–="E" THEN Kreis
GOTO start
Kreis:
CLS
IF ke–="K" THEN
  PRINT  "Kreis=K  Kreisbogen=B  Kreisausschnitt=A "
  verh!=.52
ELSE
  PRINT  "Ellipse=K  Ellipsenbogen=B  Ellipsenausschnitt=A "
END IF
kr–=UCASE–(INPUT–(1))
IF kr–="K" THEN
  IF ke–="K" THEN
    INPUT "Radius: (max.18Ø) ",r
  ELSE
    INPUT "Breit(max.18Ø), Hoehe: (max.18Ø) ",r,h
    IF h>r THEN SWAP r,h
    verh!=h/r*.52
  END IF
  IF r<181 AND r>Ø THEN CIRCLE (x1,y1),r,1,,,verh!
END IF
IF kr– ="B" OR kr–="A" THEN
  IF ke–="K" THEN
    INPUT "Radius (max 18Ø), Winkel (Ø-359 Grad) ",r,w
  ELSE
    INPUT "Breit(max.18Ø), Hoehe: (max.18Ø),
**             Winkel (Ø-36Ø Grad)  ",r,h,w
    IF h>r THEN SWAP r,h
    verh!=h/r*.52
  END IF'
  wk!=w*pi!/18Ø
  IF r<181 AND r>Ø AND w<36Ø AND w>Ø THEN
    IF kr–="B" THEN
      CIRCLE (x1,y1),r,1,Ø,wk!,verh!
    ELSE
      CIRCLE (x1,y1),r,1,-.ØØ1,-wk!,verh!
    END IF
```

```
  END IF
END IF
PRINT "Bitte Taste druecken "
taste:ta-=INKEY-:IF ta-="" THEN taste
GOTO start

ende:
END
```

Die Variablen *x1* und *y1* markieren den Mittelpunkt des Fensters. Die Funktion werden wir im nächsten Kapitel besprechen. Die Eingabe durch den Anwender wird mit INPUT$(1) an die Variable *ke$* übergeben. Diese Funktion ist für diesen Einsatzfall die kürzeste und komfortabelste. Probieren Sie ruhig eine andere Möglichkeit aus. Die zweite Eingabe durch den User erfolgt beim Label »Kreis«, abhängig von der ersten Eingabe. Es folgt die dritte Eingabe, bei der die einzelnen Parameter übergeben werden. Dabei wird in der Variablen *verh!* das Verhältnis von der Breite zur Höhe der Ellipse berechnet. Am einfachsten ist ein voller Kreis oder eine komplette Ellipse:

```
CIRCLE (x1,y1),r,1,,,verh!
```

x1 und *y1* markieren den Mittelpunkt der Ellipse. Die Variable *r* erhält den Wert für den Radius der Hauptachse in horizontaler Richtung. Handelt es sich um die Sonderform Kreis einer Ellipse, könnte die Eingabe mit *r* beendet werden. Da aber das Verhältnis Breite zu Höhe eines Pixels von Monitor zu Monitor verschieden sein kann, gleicht der Kreis dann in den meisten Fällen einem Fußball, dem etwas die Luft ausgegangen ist. Mit dem Verhältnis verh! = .52 werden Sie wahrscheinlich einen Kreis erhalten. Eventuell müssen Sie etwas experimentieren. Im Programm wird das Verhältnis für eine Ellipse berechnet, indem die beiden Radien dividiert und mit der Monitorkonstante .52 multipliziert werden. Die Formel

```
verh!=h/r*.52
```

sieht zwar so aus, als würde in h der doppelte Wert eingesetzt. Da beim Hires-Modus aber Pixelbreite und Pixelhöhe etwa ein Verhältnis von 1:2 bilden, wurde die Formel um den Faktor 2 gekürzt.

Kommen wir nun zu den Kreis- bzw. Ellipsenbogen. Dazu benötigen wir die Werte für Start und Ende des Bogens. Diese Werte stellen Winkelangaben im Bogenmaß dar und liegen im Bereich von 0 bis 2*pi. Die Umrechnung im Programm übernimmt die Programmzeile

```
wk!=w*pi!/18Ø
```

Mit diesem Wert kann nun der Bogen gezeichnet werden:

```
CIRCLE (x1,y1),r,1,Ø,wk!,verh!
```

Das Programm setzt den Wert für Start immer auf Null. Die Bögen beginnen also an der gleichen Position. Nun kommt ein kleiner Bonbon, der in einigen Büchern bei der Beschreibung der CIRCLE-Anweisung fehlt. Indem Sie die Werte für Start und Ende des Bogens negativ einsetzen, erhalten Sie keine Kreisbögen, sondern Ellipsen- bzw. Kreisausschnitte!

```
CIRCLE (x1,y1),r,1,-.ØØ1,-wk!,verh!
```

Sie brauchen also nicht mühsam die Positionen von Start und Ende berechnen, um eine Linie zum Mittelpunkt ziehen zu können. Eine Tortengrafik ist damit ein Kinderspiel. Zum Schluß des Kapitels demonstriert Ihnen ein kleines Programm recht eindrucksvoll, wie Sie mit der CIRCLE-Anweisung Spiralen zeichnen können.

```
REM Spirale  Pfad: Grundlagen/1AllerAnfang/Spirale
'P1-6
PALETTE Ø,.6,.6,.8:PALETTE 1,Ø,1,Ø
y=9Ø:a=-1
WHILE a
  CLS:RANDOMIZE TIMER
  r=1:fakt=RND*2+.1:x=RND*26Ø+18Ø:fa=RND*2+1
  WHILE r<18Ø
    FOR an=Ø TO 6.28 STEP .314
      r=r+fakt
      CIRCLE (x,y),r,fa,an,an+.314,.5
    NEXT
  WEND
IF INKEY-<>"" THEN a=Ø
WEND
```

Der Trick zum Zeichnen einer Spirale besteht darin, daß immer nur ein kleiner Kreisausschnitt gezeichnet wird, der Radius erhöht wird, wieder ein Kreisausschnitt gezeichnet und abermals der Radius vergrößert wird, etc.

Kapitel 2

Screens und Windows

2.1 Screens

Eine der grundlegenden Eigenschaften des Amiga ist die Möglichkeit, eigene virtuelle Bildschirme, genannt Screens zu programmieren. Damit kann sich jeder Programmierer den maßgeschneiderten Bildschirm für sein Programm erstellen. Bei einem Grafikprogramm wird man zum Beispiel möglichst viele Farben bereitstellen. Bei Textprogrammen soll möglichst viel Text in einer Zeile dargestellt werden. Dafür kommt man unter Umständen mit 2 Farben aus. Die Basic-Anweisung hat folgendes Format:

```
SCREEN n,Breite,Höhe,Tiefe,Modus
```

Amiga-Basic arbeitet mit einer Kenn-Nummer *n* für den Schirm. Da maximal 4 Bildschirme geöffnet werden können, stehen dafür die Werte 1 bis 4 zur Verfügung. Die Kennung wird bei der WINDOW-Anweisung benötigt, um ein Fenster einem bestimmten Schirm zuweisen zu können.

2.1.1 Darstellbare Farben

Für jeden neuen Screen wird ein bestimmter Speicherbereich im RAM reserviert. Dieser Speicher wird softwaremäßig wie ein rechteckiger Bereich in Screen-Größe (oder größer) behandelt. In diesem Bereich wird für jedes Pixel des Bildschirmes ein Bit gespeichert. Man spricht deshalb von einer Bitmap, einer Speicher-Landkarte. Diese Bitmap wiederum besteht aus einzelnen Ebenen, den Bitplanes. Je nach Anzahl der darstellbaren Farben werden 1 bis 5 solcher Bitplanes benötigt. Der Parameter *Tiefe* der SCREEN-Anweisung legt die Anzahl der Bit-Ebenen fest.

Bei nur einer Bitplane können nur zwei Farben dargestellt werden. Entweder ist ein Bit gesetzt oder nicht gesetzt. Stellt man sich zwei Ebenen übereinander vor, ergeben sich bereits vier verschiedene Bit-Kombinationen, also auch vier verschiedene Farben. Die darstellbaren Farben lassen sich aus folgender Formel berechnen:

```
2 ^ Anzahl der Bit-Ebenen
```

Tiefe (Bit-Ebenen)	darstellbare Farben
1	2
2	4
3	8
4	16
5	32

Tabellenform

Je mehr Farben Sie darstellen wollen, um so mehr Bitplanes benötigen Sie also. Damit erhöht sich natürlich auch der Speicherbedarf. Da für jeden Bildpunkt ein Bit des Speichers reserviert wird, kann man den Speicherbedarf ganz leicht selbst ausrechnen. Wichtig ist dabei nur noch, daß eine Grafikzeile in Wortlänge (16 Bit) gespeichert wird. Die Formel für den Speicherbedarf eines Screens sieht dann so aus:

```
Speicherbedarf = 2*INT((Breite+15)/16)*Höhe*Tiefe
```

Für einen Bildschirm von 640 Pixel Breite, 256 Zeilen Höhe und einer Tiefe 4 benötigen Sie

```
2*INT((640+15)/16)*256*4 = 81920 Byte.
```

2.1.2 Die Bildschirm-Modi

Kommen wir nun zu den letzten drei Parametern der SCREEN-Anweisung. Die Werte für Breite, Höhe und Modus hängen direkt voneinander ab. In der folgenden Tabelle sind die wichtigsten Gesichtspunkte übersichtlich zusammengefaßt:

Modus	Bezeichnung	Breite Pixel maximal	Höhe Pixel maximal	Tiefe Bit-planes	Speicherbedarf in Bytes
1	LoRes	320	256	1 bis 5	10240 bis 51200
2	HiRes	640	256	1 bis 4	20480 bis 81920
3	LoRes-Lace	320	512	1 bis 5	20480 bis 102400
4	HiRes-Lace	640	512	1 bis 4	40960 bis 163840
*	ExtraHalfBrite	320	256	6	61440
*	Dual PlayField	320	256	2 bis 6	10240 bis 61440
*	Hold and Modify	320	256	5 bis 6	51200 bis 61440

Die letzten drei Modi, die mit einem Stern versehen sind, haben keine Modus-Nummer. Sie wurden der Vollständigkeit halber mit aufgeführt und können vom Standard-Basic heraus nicht programmiert werden. Sie brauchen auf diese interessanten Modi jedoch nicht verzichten. Im dritten Teil des Buches erfahren Sie, wie diese doch in Basic programmiert werden können.

Aber auch die verbleibenden vier Modi bieten allerhand. Zusammen mit den verschiedenen Werten für die Tiefe können Sie 18 verschiedene SCREENS programmieren. Die ersten beiden Modi stehen für die horizontalen Bildschirmauflösungen. Der Modus 1 wird als Modus mit niedriger Auflösung (Low Resolution) bezeichnet. In der Breite stehen maximal 320 Bildpunkte zur Verfügung. Dafür können mit 5 Bitebenen 32 Farben gleichzeitig dargestellt werden. Der zweite Modus bringt mit 640 Bildpunkten pro Zeile die höchste Auflösung (High Resolution). Es können bis zu 16 verschiedene Farben gezeigt werden.

Die Modi 3 und 4 arbeiten im Interlaced-Modus. Dabei wird bei jedem Rasterstrahlendurchlauf nur ein halbes Bild im Abstand von einer Zeile gezeichnet. Damit verdoppelt sich die vertikale Auflösung. Ein Bild wird also erst nach zwei Rasterstrahlendurchläufen komplett gezeichnet. Das führt leider dazu, daß das Bild etwas flimmert. Es stehen die beiden vertikalen Auflösungen LoRes (Modus 3) und HiRes (Modus 4) zur Verfügung. Beachten Sie, daß sich mit der Verdoppelung der Auflösung in der Y-Achse auch der Speicherbedarf verdoppelt.

2.1.3 Verhältnis

Einen wichtigen Punkt für die grafische Darstellung bei den verschiedenen Bildschirm-Modi haben wir bisher nur kurz gestreift. Das richtige Verhältnis der Breite zur Höhe eines Pixels ist für eine ordentliche Grafik unerläßlich. Die folgende Tabelle zeigt Ihnen die vier Möglichkeiten:

Breite	**Höhe**	**x =**	**y =**
320	256	x=0.95 *y	y=x*1.053
640	256	x=1.9 *y	y=x*0.526
320	512	x=0.475*y	y=x*2.105
640	512	x=0.95 *y	y=x*1.053

Die in der Tabelle genannten Werte stimmen auf meinem Monitor, Commodore Modell 1081. Es kann durchaus sein, daß auf Ihrem Gerät die Werte geringfügig abweichen. Gegebenenfalls müssen Sie die Werte etwas korrigieren. – Wollen Sie zum Beispiel ein Quadrat von 100 Pixel (x) Kantenlänge zeichnen, das bei den Koordinaten 10/10 beginnt, so geben Sie ein:

```
LoRes          LINE (10,10)-(10+100,10+105),,b
HiRes          LINE (10,10)-(10+100,10+53),,b
LoRes-Lace     LINE (10,10)-(10+100,10+210),,b
HiRes-Lace     LINE (10,10)-(10+100,10+105),,b
```

2.1.4 18 auf einen Streich

Das folgende Demonstrationsprogramm zeigt Ihnen alle 18 verschiedenen Bildschirmdarstellungen. Neben dem Modus wird der freie Systemspeicher, die Anzahl der Bitmaps und die Anzahl darstellbarer Farben ausgegeben. Es werden Rechtecke in den jeweils möglichen Farben gezeichnet. Mit einem Druck auf eine beliebige Taste gelangen Sie zur nächsten Darstellung.

```
REM Screen  Pfad: Grundlagen/2ScreenWindow/Screen
'P2-1 **Beispiel für SCREEN und WINDOW**
n=1
text-="Speicher zu klein! Bitte alle Fenster schliessen."
mem=FRE(-1)
IF mem<190000& THEN PRINT text-:BEEP:END
DEFINT a-z:n=1
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):sh2=sh*2
FOR modus = 1 TO 4
  IF modus>2 THEN
    h=sh2:t-="Interlaced"
  ELSE
    h=sh
    IF modus=1 THEN t-="LowResolution" ELSE t-="HighResolution"
  END IF'
  IF modus=1 OR modus =3 THEN
    b=320:t=5
  ELSE
    b=640:t=4
  END IF
FOR tiefe = 1 TO t
  n=n+1
  SCREEN 1,b,h,tiefe,modus
  WINDOW n,,,0,1
  farben=2^tiefe
  PRINT "Aufloesung bzw. Modus="modus;t-
  PRINT "Freier Systemspeicher:       "FRE(-1)
  PRINT "Anzahl der Bitmaps (=Tiefe): "tiefe
```

```
  PRINT "Anzahl darstellbarer Farben: "farben
  breite=(WINDOW(2)-1Ø)/farben
  FOR f=Ø TO farben-1
    LINE (f*breite,5Ø)-(f*breite+breite,window(3)),f,bf
  NEXT
  taste:ta$=INKEY$:IF ta$="" THEN taste
  WINDOW CLOSE n
  SCREEN CLOSE 1
  FOR i=1 TO 1ØØØ:NEXT
NEXT tiefe,modus
END
```

Zum Programmablauf ist nicht viel zu sagen, da das Wesentliche bereits vorher besprochen wurde. Wegen des hohen Speicherbedarfs im Interlaced-Modus wird zu Beginn der freie Speicher mit der FRE-Funktion abgefragt und gegebenenfalls das Programm abgebrochen. Mit der Anweisung

```
SCREEN CLOSE n
```

wird ein zuvor mit dem Parameter *n* geöffneter Bildschirm wieder geschlossen. Eine Programmzeile wird Ihnen wahrscheinlich etwas spanisch vorkommen. Mit

```
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
```

wird die Höhe des Workbench-Screens gelesen. Mit dem ermittelten Wert (*sh*) läuft das Programm auch auf einem Amiga mit geringerer Screen-Höhe (amerikanische Norm). Sie werden dieser Zeile noch oft begegnen. Wie Sie mühelos an solche Werte gelangen, erfahren Sie etwas später bei den System-Routinen.

Fassen wir kurz das zusammen, was wir mit der SCREEN-Anweisung alles anfangen können. Sie können jeden der bis zu vier Bildschirme in einem anderen Modus darstellen. Die jeweilige Höhe des Screens kann innerhalb der genannten Grenzen liegen. Um Speicherplatz zu sparen, können Sie also auch einen Bildschirm von zum Beispiel 100 Reihen Höhe programmieren. Da aber die Bildschirme immer in der linken oberen Ecke beginnen, können mehrere Schirme nicht untereinander dargestellt werden. (Es sei denn, der Anwender zieht den Screen mit der Maus nach unten.) Die Breite eines Screens kann natürlich auch verringert werden. Da Ihnen dabei unter Umständen der Amiga abstürzen kann, ist diese Möglichkeit mit Vorsicht zu genießen.

2.2 Windows

Mit neuen Screens alleine können Sie noch nicht viel anfangen. Im Gegensatz zu den Routinen des Betriebssystems müssen Sie zur grafischen Darstellung unter Amiga-Basic zusätzlich ein oder mehrere Fenster öffnen. Mit diesen Fenstern haben Sie ein vielseitiges Werkzeug in der Hand. Sie können damit in mehreren Fenstern verschiedene Programme ablaufen lassen. Für Mitteilungen an den Anwender öffnen Sie einfach kurzzeitig ein neues Fenster. Grafische Anweisungen können Sie in einem nicht sichtbaren Fenster arbeiten lassen.

Mit der folgenden WINDOW-Anweisung erzeugen Sie ein neues Fenster, bringen es in den Vordergrund des Screens und aktivieren es. Damit leiten Sie alle Bildschirmausgaben (Text und Grafik) an dieses Window:

```
WINDOW Kennung <,<Titel><,<(x1,y1)-(x2,y2)><,<Typ><,Screen>>>>
```

Um das Fenster im Programm ansprechen zu können, erhält es eine Kennung. Der Wert muß größer als Null sein. Wollen Sie das Basic-Ausgabefenster beibehalten (es hat den Wert 1), wählen Sie für die Kennung einen Wert > 1.

Die Option *Titel* gestattet es, jeden beliebigen Text in der Titelleiste auszugeben. Dafür können Sie eine String-Variable oder direkt einen Text in Anführungszeichen einsetzen. Verzichten Sie auf diese Option, so vergrößern Sie das Fenster um die Höhe der Titelleiste.

Die Parameter *x1, y1* sind die Koordinaten der linken oberen Ecke und *x2, y2* die Koordinaten der rechten unteren Ecke des Fensters. Bei Screen geben Sie die Kennung des Screens ein, zu dem das Fenster gehört.

Überschreiten Sie mit den Fenster-Koordinaten dessen maximale Werte, so wird Ihr Programm mit einer Fehlermeldung abgebrochen. Der einfachste Weg, das größtmögliche Fenster zu programmieren ist, einfach die Koordinaten wegzulassen:

```
WINDOW 2,,,Ø,2
```

2.2.1 Die Fenster-Typen

Zur Komplettierung der WINDOW-Anweisung fehlt uns nur noch der Parameter *Typ*. Damit können Sie den Refresh-Modus und die Standard-Gadgets für Ihr Fenster bestimmen. Der Wert für *Typ* setzt sich aus den Werten der gesetzten Bits nach folgender Tabelle zusammen:

Bit	Wert	Funktion
0	1	Größen-Gadget
1	2	Verschiebe-Gadget
2	4	Tiefen-Gadget
3	8	Schließ-Gadget
4	16	Window-Refresh

Wollen Sie zum Beispiel ein Fenster, das in der Größe verändert und geschlossen werden kann, so geben Sie den Wert 9 (1+8) für den Typ an. Was bedeutet nun das Refresh? Der Amiga kennt drei Arten von Refresh-Techniken. Für das normale Basic-Fenster ist jedoch nur die Technik des Smart-Refresh interessant. Ist dieses Bit für ein Fenster gesetzt und wird dieses Fenster von einem anderen Fenster verdeckt, so werden die überlappenden Fensterteile in einen Zwischenspeicher gelegt, bevor sie verdeckt werden. Wird nun ein darüberliegendes Fenster zur Seite geschoben oder geschlossen, so wird der freiwerdende Bereich aus dem Zwischenspeicher neu gezeichnet. Damit es dabei während des Programmablaufes nicht zu Speicherproblemen kommt, reserviert das Amiga-Basic die maximale Speichergröße. Der Wert 16 reserviert also den Speicherplatz für die Fenstergröße. Kommt der Wert 1 für das Vergrößerungs-Gadget hinzu, so muß ein Speicher von der kompletten Bildschirmgröße reserviert werden. Damit können Sie schnell die Speicherkapazität Ihres Amigas erschöpfen. In den folgenden Kapiteln finden Sie genügend Beispiele zur WINDOW-Anweisung.

2.2.2 Die Brücke zum System

Nun kommen wir zur wichtigsten Funktion des Amiga-Basic. Ohne sie könnten wir viele der System-Routinen nicht benutzen. Die Informationen liefert die WINDOW-Funktion:

```
v=WINDOW(n)
```

Für den Parameter *n* setzen Sie einen Wert von 0 bis 8 ein. Die Informationen, die Sie von der Funktion zurückerhalten, sind in der folgenden Übersicht zusammengestellt:

n	**erhaltene Information (v)**
0	Die Nummer des selektierten Fensters.
1	Die Nummer des aktuellen Fensters, in dem die Ausgaben wirken.
2	Die Breite des aktuellen Ausgabefensters.
3	Die Höhe des aktuellen Fensters in Pixeln.
4	Die X-Koordinate des Text-Cursors.
5	Die Y-Koordinate des Text-Cursors.
6	Die Anzahl der erlaubten Farben des aktuellen Windows.
7	Zeiger auf die Window-Struktur.
8	Zeiger auf die Rastport-Struktur.

Wie können Sie nun diese Informationen verwerten? Probieren Sie folgende Zeile im Direktmodus:

```
PSET(WINDOW(2)/2,WINDOW(3)/2)
```

Exakt in der Mitte des Fensters erhalten Sie einen weißen Punkt. Diese Zeile können Sie in jedem Fenster von beliebiger Größe einsetzen. Der Punkt wird immer in der Mitte ausgegeben. Wenn Sie ein Unterprogramm schreiben, das unabhängig von der Fenstergröße arbeiten soll, kommen Sie ohne diese Werte nicht aus. Ein solches Unterprogramm könnte zum Beispiel eine Hardcopy-Routine sein. Auch wenn Sie Programme schreiben wollen, die auf einem Amiga mit einer anderen Bildschirmhöhe (z.B. amerikanische Version, oder Version der ersten Amigas) laufen sollen, kommen Sie um diese Abfrage nicht herum. Sie werden feststellen, daß die meisten Programme dieses Buches die unterschiedlichen Bildschirmgrößen berücksichtigen.

Die X- und Y-Koordinaten des Text-Cursors können Sie verwenden, um Grafik und Text ordentlich zu gestalten. Der Aufbau von optisch ansprechenden Eingabemasken wird damit ein Kinderspiel. Einen kleinen Einblick in die möglichen Anwendungen sehen Sie aus der folgenden Programmzeile:

```
PRINT :PRINT "Nummer:";:LINE(WINDOW(4),WINDOW(5))-
**                     (WINDOW(4)+6Ø,WINDOW(5)-8),3,bf
```

Die Funktion WINDOW(6) liefert die maximal für das aktuelle Fenster erlaubten Farben. Richtiger gesagt, erhalten Sie mit der Funktion die höchstmögliche, letzte Farbnummer. Wenn Sie die Zeile

```
PRINT WINDOW(6)
```

im Direktmodus eingeben, erhalten Sie den Wert 3 zurück, damit können 4 Farben dargestellt werden. In dem folgenden Programm sind die Funktionen WINDOW(0) bis WINDOW(6) eingearbeitet:

```
REM Fenster-Werte  Pfad: Grundlagen/2ScreenWindow/FenstWert
'P2-2 **Beispiel für WINDOW-Funktion**
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14) :wh=sh-20
SCREEN 1,640,sh,3,2
WINDOW 2,"Fenster Nummer 2",(0,0)-(160,wh),1,1
WINDOW 3,"Fenster Nummer 3",(320,0)-(480,wh),1,1
a=-1:alt = WINDOW(0)
WHILE a
  IF WINDOW(0)<> alt THEN
    GOSUB zeigen
    alt=WINDOW(0)
  END IF
  x1=WINDOW(2)*RND:x2=WINDOW(2)*RND
  y1=50+RND*(WINDOW(3)-50)
  y2=50+RND*(WINDOW(3)-50)
  LINE (x1,y1)-(x2,y2),fa
  fa=fa+1:IF fa>7 THEN fa=1
  ta-=INKEY-:IF ta-<>"" THEN a=0
WEND
WINDOW CLOSE 3
WINDOW CLOSE 2
SCREEN CLOSE 1
END

zeigen:
IF WINDOW(0)=0 THEN RETURN 'Mausklick neben Fenster
CLS
WINDOW OUTPUT WINDOW(0)
PRINT "gewaehltes Fenster"WINDOW(0)
PRINT "Fensterbreite "WINDOW(2)
PRINT "Fensterhoehe: "WINDOW(3)
PRINT "Position X:   "WINDOW(4)
PRINT "Position Y:   "WINDOW(5)
PRINT "Anzahl Farben:"WINDOW(6)+1
IF WINDOW(0) = 2 THEN
  WINDOW OUTPUT 3
ELSE
```

```
  WINDOW OUTPUT 2
END IF
CLS
PRINT "aktuelles Fenster"WINDOW(1)
RETURN
```

Zuerst wird ein Bildschirm hoher Auflösung mit einer Tiefe 3 für 8 Farben geöffnet. Anschließend werden für den Bildschirm die Fenster 2 und 3 geöffnet. Damit der Anwender die WINDOW-Funktionen besser verfolgen kann, erhalten die Fenster die jeweilige Kennung im Titel. Das Programm läuft in der WHILE/WEND-Schleife ab.

In der Schleife wird zuerst geprüft, ob der User mit einem Mausklick ein Fenster selektiert hat, welches vorher nicht ausgewählt war. Das zuletzt gewählte Fenster wird in der Variablen *alt* festgehalten. Ist *alt* von WINDOW(0) verschieden, so wurde ein anderes Fenster selektiert. Das Programm verzweigt dann zu der Subroutine »zeigen«.

In dieser Subroutine wird, um einen Programmabbruch zu verhindern, am Anfang geprüft, ob der Anwender in den Bereich ohne Fenster die Maustaste gedrückt hat. Es folgt eine weitere Form der WINDOW-Anweisung:

```
WINDOW OUTPUT Kennung
```

Das mit Kennung angesprochene Fenster wird zum aktuellen Ausgabefenster, in dem alle Ausgaben gezeigt werden. Es wird durch diese Anweisung nicht in den Vordergund geholt. Es können dadurch grafische Konstruktionen durchgeführt werden, ohne daß der Anwender sie sieht.

In unserem Programm wird das selektierte Fenster mit dieser Anweisung zum aktuellen Window. Dadurch werden die folgenden PRINT-Anweisungen in dem Fenster gezeigt. Um die richtige Anzahl der erlaubten Farben zu erhalten, wird die Funktion WINDOW(6) um eins erhöht. Anschließend wird das nicht angewählte Fenster zum aktuellen Fenster gemacht.

Wenn das Programm aus der Routine »zeigen« zurückkehrt, werden dadurch die folgenden grafischen Anweisungen in dem anderen Fenster gezeigt. Die LINE-Anweisung holt die Anfangs- und Endkoordinaten der Linie mit der RND-Funktion aus der Breite und Höhe des aktuellen Fensters. Die richtigen Werte für die Breite und Höhe des jeweiligen Fensters werden dabei mit den WINDOW-Funktionen ermittelt.

Fassen wir den Programmablauf nochmals kurz zusammen. Klickt der Anwender zum Beispiel in das Window 3, so werden dort die Fensterdaten ausgegeben. Die LINE-Anweisungen wirken dagegen im Fenster 2. Mit einem Tastendruck wird das Programm beendet. Zum Schluß werden die Fenster mit der dritten Form der WINDOW-Anweisung wieder geschlossen:

```
WINDOW CLOSE Kennung
```

Das Fenster mit der Kennung wird vom Bildschirm gelöscht. Es erscheint das Fenster, das vorher das aktuelle war. In dem Programm erscheint also das Basic-Ausgabefenster. Haben Sie das Basic-Ausgabefenster in Ihr Programm einbezogen, landen Sie auf der Workbench, wenn Sie dieses Fenster schließen. Das List-Fenster bleibt jedoch bestehen. Die beiden wichtigsten WINDOW-Funktionen habe ich bis zum Schluß aufgehoben. Wir werden im zweiten Teil des Buches detailliert darauf eingehen. Einen kleinen Vorgeschmack will ich Ihnen jedoch nicht vorenthalten. In der Tabelle lesen Sie, daß die Funktion

```
v=WINDOW(7)
```

auf die Window-Struktur zeigt. Damit ist gemeint, daß die Funktion die Startadresse der Struktur im Speicher zurückgibt. Diese Struktur ist eine Liste von Daten, die alle mit dem Fenster zusammenhängen. Dazu gehören die Position des Fensters auf dem Bildschirm, die Koordinaten des Mauszeigers, die maximalen und minimalen Fenstergrenzen und vieles mehr. Mit den PEEK-Anweisungen können Sie diese Daten jederzeit auslesen oder programmiert weiterverarbeiten. Die andere Funktion

```
v=WINDOW(8)
```

zeigt auf die Rastport-Struktur. Die Datenfelder der Struktur enthalten unter anderem die Zeichenfarben, den Füllmodus und Textinformationen. Über diese beiden Adressen können Sie über Zeiger an weitere wichtige Strukturen gelangen. Außerdem werden die beiden Funktionen bei vielen Routinen des Betriebssystems als Parameter eingesetzt. Nicht zu Unrecht können Sie die beiden letzten WINDOW-Funktionen als Brücke zum System bezeichnen.

Kapitel 3

Farben

Aus dem Kapitel über die Screens wissen Sie, wie bis zu 32 Farben auf den Bildschirm gebracht werden können. Allerdings sind die Standardfarben recht einfallslos. Aus den bisherigen Beispielen haben Sie gesehen, daß zum Beispiel die letzten 10 Farben nur aus verschiedenen Grautönen bestehen. Bevor wir diesem Problem zu Leibe rücken, schauen Sie sich erst einmal an, was es mit der Zeichenfarbe auf sich hat.

3.1 Allerlei Zeichenstifte

Bei den drei Anweisungen zum Zeichnen, PSET, LINE und CIRCLE können Sie jeweils mit der Option *Farbe* bestimmen, in welcher Farbe gezeichnet werden soll. Doch was machen Sie, wenn Sie einen Text in einer anderen Farbe ausgeben wollen?

Verantwortlich für die Darstellung von Text und Grafik ist die Vordergrundfarbe, auch Zeichenfarbe genannt. Diese Farbe ist beim Basic des Amiga auf die Farbe Nummer 1 voreingestellt. Es gibt aber eine Anweisung, mit der Sie der Zeichenfarbe eine beliebige Farbe zuweisen können:

```
COLOR <Zeichenstift><,Hintergrundstift>
```

Wundern Sie sich bitte nicht, warum die beiden Parameter der Anweisung andere Bezeichnungen tragen als die, die Sie eventuell kennen. Der Parameter *Hintergrundstift* wird in den Unterlagen üblicherweise als Hintergrundfarbe bezeichnet. Das ist falsch. Die Hintergrundfarbe ist die Farbe 0 und die können Sie mit dieser Anweisung nicht ändern. Bevor wir uns näher mit diesem Gesichtspunkt befassen, tippen Sie die folgende Zeile ein, die Ihnen die Farbänderung des Zeichenstiftes demonstriert:

```
a:COLOR INT(RND*(4)):LINE (Ø,Ø)-(RND*WINDOW(2),RND*WINDOW(3)):
**                                                      GOTO a
```

Die Grafik des Amiga unterscheidet drei verschiedene Zeichenstifte. Wir werden nun diese, auch mit den unterschiedlichen Bezeichnungen, etwas genauer betrachten, da sie auch bei den Routinen des Betriebssystems eine sehr wichtige Rolle spielen.

Der erste Stift ist Ihnen bereits geläufig. Es ist der Vordergrund- oder primäre Zeichenstift. Er ist für die grafische Ausgabe verantwortlich. Auch der Text erscheint in dieser Farbe. Dieser Stift wird auch A-Pen oder Fg-Pen genannt. Die Farbe dieses Stiftes ändern Sie, indem Sie den ersten Parameter der COLOR-Anweisung versorgen.

Der zweite Stift ist der Hintergrund- oder sekundäre Zeichenstift. Bei Textausgaben unterlegt er den Text mit der Farbe des Hintergrundstiftes. Außerdem ist dieser Stift für die PATTERN-Anweisung wichtig. Der Stift wird auch als B-Pen oder Bg-Pen bezeichnet. Mit dem zweiten Parameter der COLOR-Anweisung können Sie die Farbe dieses Hintergrundstiftes verändern. Die beiden besprochenen Stifte sind auch für den Zeichenmodus wichtig. Wir kommen im zweiten Teil des Buches darauf zurück.

Von dem dritten und letzten Stift haben Sie wahrscheinlich noch kaum etwas gehört. Er ist für die Randfarbe einer ausgemalten Fläche verantwortlich. Er wird daher AOl-Pen (Area Outline Pen) oder schlicht O-Pen genannt. Mit der COLOR-Anweisung können Sie ihn nicht verändern. Mit ihm werden wir etwas später ungewöhnliche Effekte realisieren.

3.2 Farbänderung

4096 verschiedene Farben kann der Amiga darstellen. Bisher haben Sie davon allerdings nicht viel gesehen. Wie kommt es eigentlich zu der Zahl 4096? Die Farben des Amiga sind in einer Tabelle gespeichert. Jeder Screen hat dafür eine eigene Tabelle. In dieser Tabelle sind für jede Farbe 3 * 4 Bit reserviert (1 Byte = 8 Bit). Jeweils 4 Bit stehen für einen Farbbestandteil. Die Farbe setzt sich aus den drei Bestandteilen rot, grün und blau zusammen. Mit 4 Bit können Sie 16 verschiedene Zustände darstellen. Bei drei Farbbestandteilen ergeben sich damit 16*16*16 = 4096 verschiedene Farbkombinationen. Mit einer einfachen Anweisung können Sie sich aus den einzelnen Farbbestandteilen jede Farbe zusammenmischen:

```
PALETTE Farbnummer,Rotanteil,Grünanteil,Blauanteil
```

Der Parameter *Farbnummer* erhält die Nummer der Farbe, die geändert werden soll. Da die Farben mit der Zahl 0 (Hintergrundfarbe) beginnen, liegt die Farbnummer zwischen 0 und 31. Die Parameter für den Farbanteil können Werte zwischen 0.00 und 1.00 annehmen. Am besten probieren Sie die Anweisung gleich aus:

```
PALETTE Ø,.73,1,Ø
```

Mit dieser Zeile erhalten Sie eine grüne Hintergrundfarbe (0) auf Ihrem Bildschirm. Natürlich hat es wenig Sinn, irgendwelche beliebigen Werte für die Farbbestandteile einzusetzen. Da nur 16 verschiedene Werte möglich sind (4 Bit!), können Sie folgende Bestandteile in der Anweisung vorgeben:

```
Ø.ØØ     Ø.Ø7     Ø.13
Ø.2Ø     Ø.27     Ø.33
Ø.4Ø     Ø.47     Ø.53
Ø.6Ø     Ø.67     Ø.73
Ø.8Ø     Ø.87     Ø.93     1.ØØ
```

Das folgende kleine Programm demonstriert die PALETTE-Anweisung:

```
REM FarbDemo  Pfad: Grundlagen/3Farben/FarbDemo
'P3-1 **Beispiel für PALETTE**
DEFINT a-z:sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
SCREEN 1,32Ø,sh,5,1
WINDOW 2,,,Ø,1
FOR i =1 TO 31
  COLOR i
  LOCATE i,INT(i/2)+1
  PRINT "Vordergrundfarbe:"i;
NEXT
a=-1
WHILE a
  fa=RND*31:RANDOMIZE TIMER
  PALETTE fa,RND,RND,RND
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
ende:
WINDOW CLOSE 2
SCREEN CLOSE 1
END
```

Zuerst wird ein Bildschirm mit einer Tiefe 5 für 32 Farben und das zugehörige Fenster geöffnet. Anschließend werden alle Farben in das Fenster geschrieben. In der WHILE/WEND-Schleife werden mit Hilfe von Zufallszahlen die Farbnummer und die Farbbestandteile geändert. Das bunte Feuerwerk können Sie durch einen Druck auf eine beliebige Taste beenden.

Nach dieser reinen Demonstration folgt ein praktisches Beispiel. Durch die Veränderung von einer einzigen Farbe können Sie einem statischen Bild etwas Leben einhauchen. Das Programm zeichnet eine Sonne. Die Farbe der Sonne geht von der schwarzen Hintergrundfarbe in rote Farbtöne über, ändert sich in orange und bildet zum Schluß eine gelbe Sonne. Der somit simulierte Sonnenaufgang erstreckt sich über 29 verschiedene Farbzusammensetzungen.

```
REM Morgen  Pfad: Grundlagen/3Farben/Morgen
'P3-2 **Beispiel für PALETTE**
d%=45: DIM x1%(d%),x2%(d%),y1%(d%),y2%(d%)
pi=3.141592
FOR i%=0 TO 45
  w=8*i%*pi/180:r1%=80:r2%=180
  x1%(i%)=300+(r1%*COS(w)):y1%(i%)=90+(r1%*SIN(w)/2)
  x2%(i%)=300+(r2%*COS(w)):y2%(i%)=90+(r2%*SIN(w)/2)
NEXT
FOR i%=0 TO 2:PALETTE i%,0,0,0:NEXT
CIRCLE (300,90),r1%,2,,,.475
PAINT (300,90),2
FOR i%=0 TO 45
  LINE (x1%(i%),y1%(i%))-(x2%(i%),y2%(i%)),2
NEXT i%
FOR j = 0 TO .8 STEP 1/15
  PALETTE 2,j,0,.13: CALL pause(.1)
NEXT
FOR i = 0 TO 1 STEP 1/15
  PALETTE 2,1,i,.13: CALL pause(.1)
NEXT i
CALL pause(5.5)
PALETTE 1,1,1,1
END

SUB pause (t!) STATIC
tim#=TIMER
WHILE tim#+t!>TIMER:WEND
END SUB
```

In der ersten FOR/NEXT-Schleife werden die Anfangs- und Endkoordinaten der Sonnenstrahlen berechnet. Anschließend ändert die PALETTE-Anweisung die Farbnummern 0 bis 2 in die Farbe Schwarz. Damit ist kein, das Bild störender, Fensterrahmen mehr zu sehen. Nun wird die Sonne als gefüllter Kreis (siehe nächstes Kapitel) mit den Strahlen gezeichnet. Sie sehen davon nichts, da ja die Farbnummer 2 in der Hintergrundfarbe Schwarz gezeichnet wird.

In der folgende Schleife wird der Rotwert der Farbe 2 in Stufen von 1/15 geändert. Die Farbe der Sonne geht langsam von schwarz in rot über. Die nächste Schleife verändert die Grünwerte in 16 Abstufungen. Nach einer Pause von 5,5 Sekunden wird die Zeichenfarbe auf weiß geändert, da sonst keine Schrift zu sehen wäre.

3.3 Ausmalen

Mit der Option *bf* (block fill) haben Sie bereits den ersten Befehl zum farbigen Füllen einer Fläche kennengelernt. Auf Dauer werden Sie sich kaum mit dem Ausmalen von Rechtecken begnügen. Der Amiga wäre kein Grafik-Zauberer, wenn er zum Füllen beliebiger Flächen nicht mindestens eine Anweisung bieten würde. Das Basic des Amiga stellt dafür sogar zwei Befehle bereit. Ich will Sie nicht noch länger auf die Folter spannen und Ihnen den ersten davon vorstellen:

```
PAINT <STEP>(x,y)<,Farbe,<Randfarbe>>
```

Mit der PAINT-Anweisung können Sie eine geschlossene Fläche in einer beliebigen Farbe ausmalen. Wichtig ist dabei die geschlossene Fläche. Fehlt auch nur ein Pixel in Ihrer grafischen Konstruktion, so läuft Ihnen gnadenlos das ganze Bild voll.

Die Parameter *x* und *y* stellen die Koordinaten eines Punktes dar, bei denen der Füllvorgang beginnt. Daß sich dieser Punkt innerhalb des zu füllenden Bereiches befinden muß, braucht wohl nicht besonders erwähnt zu werden. Befindet sich an dieser Stelle bereits ein Pixel in der gleichen Farbe wie die äußere Linie, so passiert gar nichts. Warum das so ist, werden Sie bei den Routinen des Betriebssystems kennenlernen.

Der Parameter *Farbe* läßt Ihnen die Wahl, mit welcher Farbnummer Sie den Bereich ausmalen wollen. Lassen Sie den Parameter weg, so tritt, wie könnte es auch anders sein, die Vordergrundfarbe in Aktion.

Beim dritten Parameter wird es interessant. Sie können damit eine andere Randfarbe wählen als die Füllfarbe. Sie erinnern sich sicherlich an den Outline-Pen aus dem Kapitel über die Zeichenstifte. Nun, hier haben Sie das erste praktische Beispiel für die Randfarbe. Als Randfarbe müssen Sie die gleiche Farbe einsetzen, mit der Sie vorher die Grafik konstruiert haben. Die Außenlinie muß dabei mit einer einheitlichen Farbnummer gezeichnet sein. Ohne den Parameter *Randfarbe* wird die Randfarbe gleich dem Parameter *Farbe*. Es folgt der gewohnte Einzeiler für das Füllen ohne eigene Randfarbe:

```
LINE (0,0)-(95,0):LINE -(50,95):LINE -(0,0):PAINT (50,10)
```

Das Beispiel zeichnet und füllt ein Dreieck in der Vordergrundfarbe. Wenn Sie in dem Beispiel bei der LINE-Anweisung eine andere Farbe einsetzen als beim Ausmalen, so läuft Ihnen das Bild voll. Abhilfe schafft hier der Parameter *Randfarbe*, wie Sie an der folgenden Zeile probieren können:

```
rf=3:f=1:CIRCLE (200,90),80,rf:PAINT (200,90),f,rf
```

Sie erhalten mit diesem Beispiel einen ausgefüllten weißen Kreis mit oranger Randfarbe. Das folgende Programm zeichnet und füllt verschiedenfarbige Kreise, in einem Fenster mit und in dem anderen Fenster ohne separate Randfarbe.

```
REM malen  Pfad: Grundlagen/3Farben/malen
'P3-3 **Beispiel für PAINT**
IF FRE(-1)<14ØØØØ& THEN BEEP:PRINT "Speicher zu klein":END
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):wh=sh-2Ø
SCREEN 1,32Ø,sh,5,1
WINDOW 2,,(Ø,Ø)-(15Ø,wh),16,1
WINDOW 3,,(153,Ø)-(3Ø3,wh),16,1
a=-1:f1=3:f2=2
WHILE a
  SWAP f1,f2
  WINDOW OUTPUT f1
  c1=1+RND*3Ø
  c2=1+RND*3Ø
  r=RND*5Ø
  x=r+RND*(148-2*r)
  y=r+RND*(wh-2*r)
  CIRCLE (x,y),r,c1
  'Window 2 = Outline-Pen
  IF f1=2 THEN PAINT (x,y),c2,c1 ELSE PAINT (x,y),c1
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
WINDOW CLOSE 2
WINDOW CLOSE 3
SCREEN CLOSE 1
END
```

Zuerst öffnen wir wieder einen neuen Bildschirm für 32 Farben. Für diesen neuen Screen werden zwei nebeneinanderliegende Fenster jeweils in der halben Breite des Bildschirmes geöffnet. Am Anfang der WHILE/WEND-Schleife wird durch die SWAP-Anweisung zwischen zwei Fenstern hin- und hergeschaltet. Anschließend werden nun, mit Hilfe der RND-Funktion, zufällige Farbnummern (ohne Hintergrundfarbe), Radien und Koordinaten des Mittelpunktes für einen Kreis ermittelt. Der Kreis wird gezeichnet und je nachdem, in welchem der beiden Fenster gerade gezeichnet wird, der Parameter *Randfarbe* gesetzt oder weggelassen. Haben Sie die Nase voll von den Kreisflächen, so können Sie mit einem Druck auf irgendeine Taste das Programm beenden.

3.4 Farb-Animation

Mit der Überschrift zu diesem Kapitel werden Sie sicherlich nichts anfangen können. Animation hat doch etwas mit Bewegung zu tun! Seit wann können sich Farben denn bewegen? Sie haben natürlich recht. Aber das Verfahren, das Sie in diesem Kapitel kennenlernen werden, schafft bewegte Bilder alleine durch die Änderung von Farben. Und bei den bewegten Bildern sind wir doch wieder bei der Animation.

Das Prinzip ist eigentlich recht einfach. Man zeichnet grafische Objekte mit einer oder verschiedenen Farbnummern in eine farbige Fläche mit der gleichen Farbe. Dadurch ist der Bereich mit einer anderen Farbnummer nicht zu erkennen. Durch hintereinandergeschaltetes Umfärben der Objekte entsteht eine Bewegung auf dem Bildschirm. Sie können zum Beispiel einige Farben eines Bildes für Blitze reservieren und diese nacheinander umfärben und schon haben Sie das schönste Gewitter. Wie so etwas aussehen kann, zeigen Ihnen die folgenden Zeilen:

```
REM Bewegung  Pfad: Grundlagen/3Farben/Bewegung
'P3-4
DEFINT a-z:SCREEN 1,320,100,5,1: WINDOW 2,,,0,1
FOR i = 0 TO 30: PALETTE i,0,0,0: NEXT
FOR i=20 TO 300 STEP 10: LINE (i,20)-(i+9,30),i/10,bf:NEXT
start:FOR i= 2 TO 30
      PALETTE i,1,1,1: FOR t= i TO 100:NEXT
      PALETTE i,0,0,0: NEXT :ta-=INKEY-:IF ta-=""THEN start
WINDOW CLOSE 2: SCREEN CLOSE 1
```

Nachdem ein Bildschirm mit Fenster für 32 Farben geöffnet ist, werden (fast) alle Farben auf Schwarz gesetzt. In den Farben 2 bis 30 werden nebeneinander 28 Rechtecke gezeichnet. Diese Rechtecke werden in der Schleife nacheinander weiß eingefärbt und anschließend wieder auf die Hintergrundfarbe gesetzt. Dadurch wird der Anschein erweckt, als wenn ein Rechteck sich vom linken zum rechten Bildrand bewegen würde. Die kleine Pause in der FOR/NEXT-Schleife ist notwendig, da Sie sonst kaum etwas erkennen würden. Nachdem Sie nun das Prinzip an dem Siebenzeiler kennengelernt haben, können wir uns an ein etwas schwierigeres, dafür aber hübscheres Programm wagen:

```
REM Gluecksrad  Pfad: Grundlagen/3Farben/Glueck
'P3-5
DEFINT a-z
SCREEN 1,320,256,5,1
WINDOW 2,,,0,1
PALETTE 0,.73,1,0 'gruen
FOR i = 2 TO 31
```

```
  PALETTE i,.73,1,Ø 'gruen
NEXT
pi!=3.141592
alt!=.ØØ1 :wert=1Ø
FOR i=12 TO 36Ø STEP 12
  fa=(i/12)+1:w!=i*pi!/18Ø
  w2!=(i+85)*pi!/18Ø:r1=8Ø:r2=12Ø
  CIRCLE (15Ø,125),12Ø,fa,-alt!,-w!,1.Ø4
  x=15Ø+(1ØØ*SIN(w2!)):x2=x/8
  y=125+(1ØØ*COS(w2!)):zeile=y/8
  IF zeile>18 THEN zeile=zeile+1
  PAINT (x,y),fa
  LOCATE zeile,x2:PRINT wert
  wert=wert+1:IF wert>9 THEN wert=Ø
  alt!=w!
NEXT i
CIRCLE (15Ø,125),12Ø,1,,,1.Ø4
CIRCLE (15Ø,125),8Ø,1,,,1.Ø4
CIRCLE (15Ø,125),78,Ø,,,1.Ø4
PAINT (15Ø,125),Ø
fa=2:altfa=31
start:
a=-1
PALETTE fa,.93,.2,Ø   'rot
LOCATE 1,1:PRINT "Bitte 2 verschiedene Tasten druecken"
beginn:
ta-=INKEY-:IF ta-=""THEN beginn
t1#=TIMER
zaehlen:
tt-=INKEY-:IF tt-=ta- OR tt-="" THEN zaehlen
t2#=TIMER
LOCATE 1,1:PRINT "Bitte druecken Sie nun die Leertaste"
WHILE a
  fa=fa+1:IF fa>31 THEN fa=2
  altfa=altfa+1:IF altfa >31 THEN altfa=2
  PALETTE fa,.93,.2,Ø   'rot
  PALETTE altfa,.73,1,Ø 'gruen
  ta-=INKEY-:IF ta-=CHR-(32) THEN a=Ø
WEND
t3#=t2#-t1#
```

```
IF t3#>100 THEN dauer=t3# ELSE dauer=(t2#-t1#)*100
FOR i= 0 TO dauer
  fa=fa+1:IF fa>31 THEN fa=2
  altfa=altfa+1:IF altfa >31 THEN altfa=2
 PALETTE fa,.93,.2,0   'rot
  PALETTE altfa,.73,1,0 'gruen
  FOR n=1 TO i:NEXT
NEXT
IF fa=2 THEN
  FOR i = 1000 TO 2000 STEP 20: SOUND i,1,255: NEXT
  FOR i = 2000 TO 1000 STEP -20: SOUND i,1,255: NEXT
END IF
LOCATE 1,1:PRINT "Fuer Neustart die Leertaste druecken"
taste:ta-=INKEY-:IF ta-=""THEN taste
IF ta-= CHR-(32) THEN start
WINDOW CLOSE 2
SCREEN CLOSE 1
END
```

Zuerst öffnen wir wieder einen neuen Bildschirm mit der Tiefe 5 und ein Fenster. Anschließend werden alle Farbnummern mit Ausnahme der Vordergundfarbe (Farbe 1) grün eingefärbt. In der folgenden FOR/NEXT-Schleife wird die Grafik gezeichnet. Dazu zeichnet das Programm mit der CIRCLE-Anweisung einen Kreis, bestehend aus 30 Kreisausschnitten mit den Farbnummern 2 bis 31. Die Variablen *x* und *y* werden berechnet und mit Ihnen die PAINT-Anweisung versorgt. Die Kreisausschnitte werden also eingefärbt. Den einzelnen Kreisausschnitten werden, mit Hilfe der PRINT-Anweisung, Werte von 0 bis 9 und einem Abschnitt wird der Wert 10 zugeordnet.

Die folgenden drei CIRCLE-Anweisungen zeichnen zwei Kreise um die Zahlen und einen dritten, mittleren Kreis, der anschließend mit der Hintergrundfarbe gefüllt wird. Bei dem Label »start« beginnt das eigentliche Programm. Der Bildausschnitt mit der Farbnummer 2 wird rot eingefärbt. Der Text

```
PRINT "Bitte 2 verschiedene Tasten druecken"
```

wird ausgegeben. Der Zeitpunkt, zu dem der Anwender die beiden Tasten drückt, wird in den Variablen *t2#* und *t1#* festgehalten. In der WHILE/WEND-Schleife rotiert das Glücksrad, bis die Leertaste gedrückt wird. Dazu wird die aktuelle Farbnummer mit der PALETTE-Anweisung rot gefärbt. Die vorhergehende Farbnummer erhält wieder die Hintergrundfarbe Grün. Die Farbe wird um eins erhöht. Mit ihr wird die nächste Fläche rot dargestellt etc. Sobald die Leertaste gedrückt wird, wird *a* unwahr und die Schleife verlassen.

Die Variable *t3#* ermittelt die Zeit aus *t2#* – *t1#* und übergibt sie an die Variable *dauer.* Mit dieser Variablen wird die Anzahl der Durchläufe für die folgende FOR/NEXT-Schleife begrenzt. Die Pausen sind mit dem Schleifenzähler *i* gekoppelt, so daß sich eine langsamer werdende Bewegung ergibt. Nach Beendigung der Schleife wird geprüft, ob der Anwender den Wert 10 erreicht hat. Das ist der Fall wenn die Farbnummer 2 aktuell ist. Dieser Hauptgewinn wird mit einer Sirene angekündigt. Der Anwender kann mit einem Druck auf die Leertaste einen weiteren Durchgang starten, oder mit einer beliebigen Taste das Programm beenden.

An diesem kleinen Beispiel sehen Sie, welche eindrucksvollen Ergebnisse die Farbanimation bringen kann. Ein weiteres Beispiel finden Sie in dem Kapitel mit den Programmen. Dort wird an einem Modell der Viertaktmotor simuliert.

Kapitel 4

Flächen

Sämtliche Geraden, die wir bisher zeichneten, waren durchgezogene Linien. Muß das so sein? Alle Flächen der vorhergehenden Kapitel waren einfarbig, bzw. gleichmäßig gefüllt. Gibt es dafür eine andere Möglichkeit? Natürlich gibt es diese! Sie werden sie gleich kennenlernen. Außerdem finden Sie in diesem Abschnitt einen Befehl, der schnell, der sehr schnell beliebige Vielecke zeichnet und ebenso flink füllt. Da Sie in diesem Kapitel mit Bits und Hex-Zahlen arbeiten werden, werfen wir jedoch zuerst einen Blick auf die Welt der Zahlen des Amiga.

4.1 Ohne Zahlen geht es nicht

Böse Leute, die sicher keine Computer-Freaks sind, behaupten, daß ein Computer ein dummes Ding ist, das noch nicht einmal bis 2 zählen kann. Nun, über Dummheit kann man streiten. Richtig ist dagegen, daß ein Computer im Kern nur zwei Zustände kennt: Stromkreis an und Stromkreis aus. Demzufolge beruht seine Zahlenwelt auf den beiden Zahlen 0 und 1. Man spricht, im Gegensatz zu dem uns geläufigeren Zehnersystem, von einem dualen Zahlensystem (Zweiersystem, binäres oder dyadisches Zahlensystem). Diese kleinste Einheit wird als Bit bezeichnet. Ein Bit ist entweder gesetzt, also 1, oder nicht gesetzt, also 0. Mit zweien dieser Bits können Sie jedoch bereits 4 verschiedene Zustände oder Werte darstellen:

```
ØØ  Ø1  1Ø  11
```

Mit jedem weiteren Bit wächst die Zahl der Möglichkeiten quadratisch an. Drei Bits lassen bereits 8 Zahlen zu:

```
ØØØ ØØ1 Ø1Ø Ø11 1ØØ 1Ø1 11Ø 111
```

8 Bit bilden eine neue Einheit, ein Byte. Damit sind 256 verschiedene Zahlen, von 0 bis 255, darstellbar. Die einzelnen Bits in einem Byte weisen also folgende Werte auf:

```
Bit:    7    6    5    4    3    2    1    Ø
Wert:  2^7  2^6  2^5  2^4  2^3  2^2  2^1  2^Ø
oder:  128  64   32   16   8    4    2    1
```

Wenn Sie die Werte für die einzelnen Bits zusammenzählen, erhalten Sie wieder den Maximal-Wert 255. In dem Speicher Ihres Computers können Sie jedes einzelne Byte ansprechen. Mit einem 8-Bit-Computer wären Sie damit am Ende der Zahlendarstellung. Unser Amiga hat aber einen 16/32-Bit-Prozessor, und da geht es natürlich weiter. Die nächste Einheit faßt zwei Byte zusammen und heißt Wort. Die darstellbaren Zahlen erhöhen sich damit auf 65536. Für ein Wort können Sie sich bedingt eine kurze Ganzzahl (INT, %) vorstellen. Bedingt deshalb, weil wir in unseren Betrachtungen die negativen Zahlen außer acht lassen.

Die größte Einheit ist ein Langwort. In einem Langwort sind vier Byte oder zwei Worte zusammengefaßt. Ohne Berücksichtung des Vorzeichens entspräche diese Einheit einer langen Ganzzahl (LNG, &). Nun wäre die Darstellung solcher großen Zahlen im Dual-System wenig praktikabel. Bei einem Langwort wäre sie immerhin 32 Zahlen lang. Man faßt deshalb computergerecht jeweils vier Bits zusammen. Man nennt die resultierenden Zahlen hexadezimale Zahlen. Vier Bits oder ein Nibbel (Halbbyte) können, wie Sie erfahren haben, die Zahlen 0 bis 15 darstellen. Unser Zahlensystem kennt aber nur die Zahlen 0 bis 9. Nun, auch hier hat man sich etwas einfallen lassen, die Zahlen 10 bis 15 werden einfach durch die Buchstaben A bis F dargestellt. Zusammenhängend finden Sie einige Werte der Hexadezimalzahlen in der folgenden Tabelle.

dezimal	**hexadezimal**	**dezimal**	**hexadezimal**
1	1	10	A
2	2	11	B
3	3	12	C
4	4	13	D
5	5	14	E
6	6	15	F
7	7	16	10
8	8	17	11
9	9	65535	FFFF

Sicherlich war der kurze Ausflug in das Zahlensystem für einige Leser überflüssig. Andere dagegen werden sich erst an die neue Rechnungsweise gewöhnen müssen. Bei den folgenden Programm-Beispielen werde ich deshalb noch detailliert auf dieses Thema eingehen.

4.2 Mustergültig

Die bereits weiter oben angekündigte Möglichkeit der grafischen Darstellung von beliebigen Linienmustern oder Füllmustern benötigt nur eine einzige Anweisung. Mit ihr können Sie entscheiden, ob Sie nur die Linien, oder nur die zu füllenden Flächen, oder beides verändern wollen:

```
PATTERN <Linienmuster><,Füllmuster>
```

4.2.1 Linienmuster

Schauen wir uns zuerst den Parameter *Linienmuster* der Anweisung an. Hierfür setzen Sie eine kurze Ganzzahl (integer) ein. Den Wert ermitteln Sie aus den 16 Bit des Wortes. Dabei bedeutet ein gesetztes Bit, daß ein Bildpunkt in der Vordergrundfarbe gezeichnet wird, und ein nicht gesetztes Bit, daß an dieser Stelle die Hintergrundfarbe (unsichtbar) zum Tragen kommt. Das Ganze nennt man eine Maske.

Damit Sie nicht umständlich die gesetzten Bits in eine Dezimalzahl umrechnen müssen, setzen Sie am einfachsten dafür eine hexadezimale Zahl ein. Bei der Hexadezimalzahl steht jede Zahl oder jeder Buchstabe für vier Bits. Diese schreiben Sie in der richtigen Reihenfolge auf und fertig ist Ihre Maske. Am einfachsten erkennen Sie das an einem Beispiel:

```
X.X. X.X. X.X. X.X.
8421 8421 8421 8421
  A    A    A    A
```

Ein X bedeutet in dem Beispiel ein gesetztes Bit und ein Punkt sagt, daß an dieser Stelle kein Pixel gezeichnet wird. Das Beispiel zeichnet eine Linie, die abwechselnd aus einem gesetzten und nicht gezeichneten Punkt besteht. Die 16 Bit (Wort) breite Maske ist in 4 Nibbels (=ein Halbbyte) aufgeteilt. Unter den Nibbels finden Sie die Werte der einzelnen Bits. Die Werte stellen die Zweierpotenz von rechts nach links dar (2^0=1, 2^1=2, 2^2=4, 2^3=8). Da in jedem Nibbel die Bits 1 und 3 gesetzt sind, ergibt sich als Wert 2+8=10, oder hexadezimal=A. Die komplette Maske lautet daher &HAAAA.

Da diese Art der grafischen Darstellung Ihnen noch oft begegnen wird, schauen wir uns dazu ein zweites Beispiel an:

```
..XX ..XX ..XX ..XX
  3    3    3    3
```

Diese Maske setzen wir gleich in unsere Anweisung ein:

```
PATTERN &H3333: LINE (Ø,Ø)-(6ØØ,18Ø)
```

Sie erhalten mit diesem Beispiel eine diagonale Linie auf Ihrem Monitor. Die Linie besteht immer aus 2 gezeichneten und 2 nicht gezeichneten Bildpunkten. Sie können also mit dieser Anweisung im Rhythmus von 16 Bildpunkten jedes beliebige Linienmuster programmieren.

Haben Sie die Programmzeile ausprobiert? Nun, dann haben Sie festgestellt, daß Ihnen das Linienmuster auch nach der Ausführung der Anweisung erhalten bleibt. Solange Sie kein neues Muster eingeben, bleibt Ihnen die neue Linie erhalten. Wenn Sie den Normalzustand wieder herstellen wollen, geben Sie einfach eine Maske ein, in der alle Bits gesetzt sind:

```
PATTERN &HFFFF
```

Damit sind aber die Möglichkeiten der Linienmuster noch nicht erschöpft. Sie haben sicher schon gemerkt, daß ich kein Freund von starren Bildern bin. Versuchen wir doch ein bißchen Bewegung in die Sache zu bringen:

```
REM Rahmen  Pfad: Grundlagen/4Flaechen/Rahmen
'P4-1 **Beispiel für PATTERN,Linienmuster**
LOCATE 12,25
PRINT "ACHTUNG! Anleitung beachten!"
a=-1
WHILE a
  PATTERN &H3333                  '..XX..XX..XX..XX
  LINE (1ØØ,7Ø)-(5ØØ,11Ø),1,b
  PATTERN &H6666                  '.XX..XX..XX..XX.
  LINE (1ØØ,7Ø)-(5ØØ,11Ø),1,b
  PATTERN &HCCCC                  'XX..XX..XX..XX..
  LINE (1ØØ,7Ø)-(5ØØ,11Ø),1,b
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
PATTERN &HFFFF                    'volle Linie herstellen
LINE (1ØØ,7Ø)-(5ØØ,11Ø),1,b       'und zeichnen
```

Das kleine Programm rahmt eine Anweisung recht eindrucksvoll ein. Der rechteckige Rahmen scheint sich um die Warnung herum zu bewegen. Schuld daran sind die drei PATTERN-Anweisungen in der WHILE/WEND-Schleife. Das Linienmuster aus unserem letzten Beispiel (Punkt, Punkt, leer, leer, Punkt, Punkt, leer etc.) wird nacheinander um ein Bit nach rechts verschoben. Dadurch scheint sich die Linie zu bewegen. Wie die einzelnen Bits gesetzt sind, sehen Sie als Bemerkung hinter der jeweiligen PATTERN-Anweisung. Dabei bedeutet wieder ein X, daß das jeweilige Bit gesetzt ist, und ein Punkt, daß für dieses Bit die Hintergrundfarbe gezeigt wird. Durch einen Tastendruck wird die Schleife verlassen. Es wird die Maske für eine durchgezogene Linie programmiert und das Rechteck gezeichnet.

Wenn Sie ein ähnliches Programmsegment in Ihr eigenes Programm einbauen, ist Ihnen die Aufmerksamkeit des Anwenders gewiß. Aber nicht nur für Warnhinweise ist diese pfiffige und einfache Methode geeignet. Denkbar wäre auch die optische Unterstützung eines Ablaufschemas. Ein elektrischer Schaltkreis könnte nach der jeweiligen Schalterstellung den Stromfluß darstellen. Ihnen fallen sicher noch weitere eindrucksvolle Beispiele dazu ein.

4.2.2 Füllmuster

Mit dem zweiten Parameter der PATTERN-Anweisung können Sie ganze Flächen nach Ihrem Gutdünken gestalten. Die Programmierung ist ähnlich der des Linienmusters. Die Breite ist wieder auf 16 Bit begrenzt. Um ein Flächenmuster zu erhalten, setzen Sie mehrere Linienmuster übereinander. Das Ganze wird in einem eindimensionalen Feld gespeichert und der Name der Feldvariablen an die PATTERN-Anweisung übergeben. Die Anzahl der Grafiklinien muß immer eine Potenz von zwei sein. Es sind also Füllmuster von $2^1=2$, $2^2=4$, $2^3=8$, $2^4=16$ etc. Grafikzeilen Höhe gestattet. Probieren Sie das Gelesene gleich an einem kleinen Beispiel aus:

```
REM pat  Pfad: Grundlagen/4Flaechen/pat
'P4-2 **Beispiel für PATTERN,Flächenmuster**
DEFINT m
DIM muster (3)
muster(Ø)=&H5Ø5  '.....X.X.....X.X
muster(1)=&HAØA  '....X.X.....X.X.
muster(2)=&H5454 '.X.X.X...X.X.X..
muster(3)=&H2828 '..X.X.....X.X...
PATTERN ,muster
LINE (1Ø,1Ø)-(1ØØ,1ØØ),1,bf
ERASE muster
DIM muster(1)
muster(Ø)=&HFFFF 'volles Muster
muster(1)=&HFFFF
PATTERN ,muster  'Muster loeschen
ERASE muster
LINE (11Ø,1Ø)-(2ØØ,1ØØ),1,bf
```

Zuerst definieren wir die Variablen, die mit *m* beginnen, als kurze Ganzzahl, da für das Füllmuster Integer-Werte (in Wortgröße) benötigt werden. Nun dimensionieren wir die Feldvariable *muster()* mit 4 Feldelementen. Es folgen die einzelnen Feldelemente als hexadezimale Zahlen. Damit Sie die Konstruktion des Musters leichter nachvollziehen können, sind die gesetzten (X) und die nicht gesetzten (.) Bits wieder als Bemerkung in der gleichen Programmzeile zu finden.

Die PATTERN-Anweisung erhält anschließend das Feld für das Füllmuster. In einem kleinen Rechteck wird das Füllmuster, ein verkleinertes Amiga-Symbol, dargestellt. Der Rest des Programmes führt nur Aufräumungsarbeiten durch, damit andere Grafik-Programme nicht mit einem ungewollten Muster erscheinen. Wie in den Programm-Kommentaren zu sehen ist, setzt ein zwei Zeilen hohes Muster mit durchgezogenen Linien die erste PATTERN-Anweisung außer Kraft. Diese Art der Löschung können Sie für Muster mit beliebiger Höhe anwenden.

Nachdem das erste Beispiel so schön geklappt hat, wagen wir uns an ein 8 Grafikzeilen hohes Gebilde. Außerdem kann auch dem Füllmuster etwas Action nichts schaden:

```
REM Pfeil  Pfad: Grundlagen/4Flaechen/Pfeil
'P4-3 **Beispiel für Flächenmuster mit Animation**
DEFINT a,m
DIM a1(7),a2(7),a3(7),muster(1)
a1(0)=&HC0C0  'XX......XX......
a1(1)=&HF0F0  'XXXX....XXXX....
a1(2)=&HFCFC  'XXXXXX..XXXXXX..
a1(3)=&HFEFE  'XXXXXXX.XXXXXXX.
a1(4)=&HFCFC  'XXXXXX..XXXXXX..
a1(5)=&HF0F0  'XXXX....XXXX....
a1(6)=&HC0C0  'XX......XX......
a1(7)=&H0     '................
a2(0)=&H3030  '..XX......XX....
a2(1)=&H3C3C  '..XXXX....XXXX..
a2(2)=&H3F3F  '..XXXXXX..XXXXXX
a2(3)=&HBFBF  'X.XXXXXXX.XXXXXX
a2(4)=&H3F3F  '..XXXXXX..XXXXXX
a2(5)=&H3C3C  '..XXXX....XXXX..
a2(6)=&H3030  '..XX......XX....
a2(7)=&H0     '................
a3(0)=&HC0C   '....XX......XX..
a3(1)=&HF0F   '....XXXX....XXXX
a3(2)=&HCFCF  'XX..XXXXXX..XXXX
a3(3)=&HEFEF  'XXX.XXXXXXX.XXXX
a3(4)=&HCFCF  'XX..XXXXXX..XXXX
a3(5)=&HF0F   '....XXXX....XXXX
a3(6)=&HC0C   '....XX......XX..
a3(7)=&H0     '................
muster(0)=&HFFFF 'volles Muster
muster(1)=&HFFFF
a=-1
```

```
WHILE a
  PATTERN ,a3
  LINE (3Ø,8)-(58Ø,15),1,bf
  PATTERN ,a2
  LINE (3Ø,8)-(58Ø,15),1,bf
  PATTERN ,a1
  LINE (3Ø,8)-(58Ø,15),1,bf
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
PATTERN ,muster  'Muster loeschen
ERASE a1,a2,a3,muster
```

Dieses Mal dimensionieren wir zu Beginn die drei Feldvariablen *a1()*, *a2()* und *a3()* mit 8 Elementen und die Variable *muster()* mit zwei Feldelementen. Die Konstruktion der drei Muster *a1()*, *a2()* und *a3()* finden Sie wieder in der gewohnten Art und Weise vor. Sicher ahnen Sie bereits das Ergebnis der Animation aus den drei Bildern, die aus dem Buchstaben X bestehen.

In der WHILE/WEND-Schleife werden schnell hintereinander die drei Füllmuster in einem langen schmalen Rechteck gezeichnet. Das Ergebnis ist eine Reihe von Pfeilen, die sich scheinbar schnell vom linken zum rechten Bildrand bewegen. Der Rest des Programmes bringt nichts Neues.

Das Programm zeigte Ihnen natürlich nur eine von tausend Möglichkeiten für bewegte Füllmuster. Was halten Sie von einem Schiff, das auf bewegten Wellen dahingleitet? Was denken Sie über eine Straße, die sich scheinbar bewegt, weil das Kopfsteinpflaster ein animiertes Füllmuster ist? Ebenso können Sie Schnee, Regen oder bewegte Ausströmgase eines Triebwerkes mit einem PATTERN-Muster programmieren. Wenn Sie Lust dazu haben, können Sie also vielen Grafiken mit einer Serie einfacher Füllmuster Dynamik verleihen.

4.3 Schnellzeichner

Daß der Amiga Linien und Rechtecke mit einer atemberaubenden Geschwindigkeit zeichnet und die Rechtecke ebenso schnell füllt, das wissen Sie bereits. Daß er das mit einem beliebigen Vieleck (Polygon) auch fertigbringt, werden Sie gleich erleben. Sie brauchen dazu nur zwei Anweisungen, eine zum Zeichnen und eine zum Füllen des Polygons. Schauen Sie sich zuerst den Befehl zum Zeichnen an:

```
AREA <STEP>(x,y)
```

Mit der Anweisung erhalten Sie einen Eckpunkt des Vieleckes bei den Koordinaten *x* und *y*. Mit den Koordinaten müssen Sie innerhalb des aktuellen Fensters bleiben. Bei

der Option STEP wird der mit den Koordinaten *x* und *y* gesetzte Punkt wieder vom Ausgangspunkt des grafischen Cursors gerechnet. In unserem ersten Beispiel wenden wir aber absolute Koordinaten an:

```
AREA (Ø,Ø):AREA (1ØØ,2Ø):AREA (1ØØ,1ØØ)
```

Mit diesen drei Anweisungen zeichnen Sie ein Dreieck. Sie müssen es nicht schließen, indem Sie zum Schluß nochmals die ersten Koordinaten eingeben. Daß diese Anweisung nur die halbe Miete ist, habe ich bereits angekündigt. Zum Füllen eines mit AREA gezeichneten Vieleckes setzen Sie folgende Anweisung ein:

```
AREAFILL <Modus>
```

Der Modus legt fest, wie das Polygon ausgemalt wird. Haben Sie vorher mit einer PATTERN-Anweisung ein Muster definiert, so wird das Vieleck mit diesem Muster gefüllt. Dabei bedeutet der Modus 0 (das ist der voreingestellte Wert), daß das Vieleck normal in der Vordergrundfarbe ausgemalt wird. Der Modus 1 dagegen füllt das Polygon invertiert, bei nicht geänderten Farben ist das orange. Ergänzen wir nun unser erstes Beispiel entsprechend:

```
DEFINT a:DIM a(1):a(Ø)=&H3333: PATTERN ,a
AREA (Ø,Ø):AREA (1ØØ,2Ø):AREA (1ØØ,1ØØ):AREAFILL Ø
```

Diese beiden Zeilen zeichnen nun das Dreieck und füllen es mit dem Muster. Die erste Zeile des Musters hat dabei folgendes Aussehen:

```
..XX..XX..XX..XX
```

Da wir die zweite Zeile zwar dimensioniert, aber nicht definiert haben, erhält sie den Wert Null (*a(1)*=0). Im Muster erscheint daher eine leere Zeile. Den Normalzustand stellen Sie wie inzwischen gewohnt her:

```
a(Ø)=&HFFFF:a(1)=&HFFFF:PATTERN ,a
```

Das kurze Beispiel konnte Ihnen sicherlich nicht die versprochene atemberaubende Geschwindigkeit des Zeichnens und Füllens der beiden Anweisungen demonstrieren. Mit dem folgenden Programm löse ich mein Versprechen ein:

```
REM Sechseck  Pfad: Grundlagen/4Flaechen/Sechs
REM P4-4 **Beispiel für AREA und AREAFILL**
DEFINT a-z
SCREEN 1,32Ø,256,5,1
WINDOW 2,"bitte Taste druecken",,1,1
a=-1
WHILE a
  FOR i =Ø TO 5
```

```
    x(i)=3ØØ*RND:y(i)=23Ø*RND
    AREA (x(i),y(i))
  NEXT
 fa =31*RND:COLOR fa
  AREAFILL Ø
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
WINDOW CLOSE 2: SCREEN CLOSE 1
```

Habe ich zuviel versprochen? Sicherlich nicht! Das Programm zeichnet rasend schnell ein gefülltes Sechseck nach dem anderen. Mit einem Druck auf eine beliebige Taste können Sie die Farbenpracht beenden.

Das Programm ist schnell erklärt. Zuerst wird ein 32-Farben-Screen mit Fenster geöffnet. Das Programm läuft wieder in einer WHILE/WEND-Schleife ab. Dort werden zuerst in der FOR/NEXT-Schleife mit Zufallszahlen die X- und Y-Koordinaten eines Eckpunktes ermittelt und mit AREA gesetzt. Die Schleife wird, da wir ein Sechseck zeichnen wollen, sechs Mal durchlaufen. Die Vordergrundfarbe *fa* wird anschließend durch eine Zufallszahl ermittelt und gesetzt. Mit AREAFILL wird schließlich das Vieleck gefüllt. Wurde keine Taste gedrückt, beginnt das Spiel von vorne. Durch die Zufallszahlen ergeben sich laufend überschneidende Linien, so daß daraus verschiedene Vielecke (zum Beispiel zwei Dreiecke) entstehen, die aber alle ordentlich gefüllt werden.

Die AREA-Anweisung ist zu vielseitig, um sie mit nur einem Beispiel vorzustellen. Das nächste Programm zeigt eine weitere Facette der Möglichkeiten, die in dieser Anweisung stecken. Wie schon der Name sagt, zeichnet das Programm eine Uhr, die Ihnen die aktuelle Zeit, auf die Sekunde genau, zeigt. Voraussetzung dafür ist natürlich, daß Sie die innere Uhr des Amiga richtig gestellt haben. Die AREA-Anweisung zeichnet die Zeiger so schnell, daß Sie das ständige Neuzeichnen der Zeiger nicht bemerken. Die Fenstergröße darf während des Programmablaufes beliebig verändert werden.

```
REM Clock  Pfad: Grundlagen/4Flaechen/Clock
REM P4-5 **Beispiel für AREA und AREAFILL**
ON TIMER(1) GOSUB zeit
DEFINT a-z:d=6Ø
DIM SHARED x1(d),x2(d),y1(d),y2(d),x3(d),y3(d)
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
SCREEN 1,32Ø,sh,3,1
WINDOW 2,"ENDE = Taste druecken",,1,1
start:
CLS:x=WINDOW(2)/2:y=WINDOW(3)/2  'Mittelpunkt
IF y>x THEN y=x         'Fenstergrenze schuetzen
```

```
r1=y/2                  'Radius f. Zeigerverbreiterung
r2=y/4*3                'Radius f. Zeigerspitze
r3=y/2Ø*19              'Radius f.Innenkreis
pi!=3.141592
strich=INT((y-r3)/2)    'Strichstaerke
FOR sek = Ø TO 59
  winkel=sek-15         'Ausgleich Uhr zu Bogenbeginn
  w1!=winkel*6*pi!/18Ø  'Zeigerspitze
  w2!=w1!-.Ø6           'linke Seite
  w3!=w1!+.Ø6           'rechte Seite
  x1(sek)=x+(r2*COS(w1!)):y1(sek)=y+(r2*SIN(w1!))
  x2(sek)=x+(r1*COS(w2!)):y2(sek)=y+(r1*SIN(w2!))
  x3(sek)=x+(r1*COS(w3!)):y3(sek)=y+(r1*SIN(w3!))
NEXT sek
CIRCLE (x,y),y,3,,,1.Ø53  :PAINT(x,y),3
CIRCLE (x,y),r3,Ø,,,1.Ø53 :PAINT(x,y),Ø
LINE (x-r3,y-strich)-(x-r2,y+strich),3,bf
LINE (x+r3,y-strich)-(x+r2,y+strich),3,bf
LINE (x-strich,Ø)-(x+strich,y-r2),3,bf
LINE (x-strich,y+y)-(x+strich,y+r2),3,bf
wb=WINDOW(2):wh=WINDOW(3)
TIMER ON
a=-1:bb=-1:b=-1
WHILE a
  ta–=INKEY–:IF ta–<>"" THEN a=Ø:bb=Ø
WEND
TIMER OFF
IF bb THEN start
WINDOW CLOSE 2: SCREEN CLOSE 1
END

zeit:
IF wb<>WINDOW(2) OR wh<>WINDOW(3) THEN a=Ø:RETURN
h=VAL(LEFT–(TIME–,2)):IF h>11 THEN h=h-12
m=VAL(MID–(TIME–,4,2))
s=VAL(RIGHT–(TIME–,2))
h=h*5
  COLOR Ø
  CALL Zeiger(vs)
  IF vm<>m OR vs=m THEN CALL Zeiger(vm)
  IF vh<>h OR vs=h THEN CALL Zeiger(vh)
  COLOR 4: CALL Zeiger(s)
```

```
  IF b THEN     'h- und m-Zeiger restaurieren
    COLOR 5: CALL Zeiger(m)
    COLOR 6: CALL Zeiger(h)
    b=Ø
  END IF
  IF vm<>m OR vs= m THEN COLOR 5: CALL Zeiger(m):b=-1
  IF vh<>h OR vs= h THEN COLOR 6: CALL Zeiger(h):b=-1
  vs=s:vm=m:vh=h
RETURN
SUB Zeiger(s%) STATIC
  SHARED x,y
  AREA (x,y)
  AREA (x2(s%),y2(s%))
  AREA (x1(s%),y1(s%))
  AREA (x3(s%),y3(s%))
  AREAFILL Ø
END SUB
```

Zu Programmbeginn setzen wir mit der ON TIMER-Anweisung eine Unterbrechungsreaktion zum Label »zeit«, die jede Sekunde aktiv wird. Nach dem Öffnen eines neuen Screens mit Window erhalten die einzelnen Variablen ihre Werte zugewiesen. Das Programm ist so ausgelegt, daß es bei jeder Fenstergröße arbeitet. Alle Variablen basieren auf den beiden Veränderlichen *x* und *y*, die wiederum auf den Fensterabmessungen beruhen.

In der FOR/NEXT-Schleife werden die einzelnen Werte den sechs Feldvariablen zugewiesen. Jedes Variablen-Feld steht für eine Ecke des Zeigers, bei einer von 60 Zeigerstellungen. Anschließend wird das Zifferblatt gezeichnet. Bevor der Timer zu arbeiten beginnt, werden noch schnell die aktuellen Fensterabmessungen in den beiden Variablen *wb* und *wh* notiert. In der WHILE/WEND-Schleife wartet das Programm auf einen Tastendruck, der das Programm beenden würde.

Wie schon zu Beginn erwähnt, wird alle Sekunde die Subroutine »zeit« angesprungen. Dort wird zuerst überprüft, ob inzwischen die Fenstergröße verändert wurde. In diesem Fall wird die Schleifenvariable *a* unwahr und die WHILE/WEND-Schleife verlassen. Hat sich dagegen nichts verändert, so werden aus der TIME$-Systemvariablen die aktuelle Stunde (*h*), Minute (*m*) und Sekunde geholt (*s*). Die Stunde wird mit 5 multipliziert, damit sie in die 60-er Zeigeranzeige paßt. Nun wird der alte Sekundenzeiger mit COLOR 0 gelöscht. Sollte sich bei den Minuten oder Stunden inzwischen ein neuer Wert ergeben, werden diese Zeiger ebenfalls gelöscht. Anschließend werden alle gelöschten Zeiger neu gezeichnet. Ein Überschneiden der Zeiger wird ebenfalls berücksichtigt. Am Ende der Subroutine werden die aktuellen Zeigerstellungen für den nächsten Aufruf in Variablen gespeichert.

Das Zeichnen der einzelnen Zeiger übernimmt das Unterprogramm »Zeiger«. Da die Feldvariablen zu Programmbeginn mit DIM SHARED als globale Variablen deklariert wurden, gelten sie auch für das Unterprogramm. Das war schon alles. Ist das nicht toll, wie schnell die AREA-Anweisung die Zeiger zeichnet? Sicherlich haben Sie durch die gezeigten Beispiele genug Anregungen für eigene Programme erhalten. Ja? Dann nichts wie los und die eigenen Ideen in effektvolle Programme umsetzen.

Kapitel 5

Geschäftsgrafik

Wenn man im Geschäftsleben die Entwicklung, die eine Sache genommen hat, verfolgen will, so bedient man sich meistens der grafischen Darstellung. Sie ist in der Regel einfacher zu erfassen als trockenes Zahlenmaterial. Beispiele dafür sind Grafiken über Verkaufs- oder Umsatzzahlen der einzelnen Monate, Preisentwicklungen, Personalentwicklungen, ABC-Analysen und vieles mehr. Dieses Kapitel ist diesem Thema gewidmet. Auch im privaten oder schulischen Bereich gibt es dafür eine Reihe von Anwendungsmöglichkeiten. So können Sie mit einer kleinen Grafik hervorragend den Benzinverbrauch Ihres Fahrzeuges, bzw. dessen gesamte Kostenentwicklung verfolgen. Unser Umweltbewußtsein wird immer ausgeprägter. Um mit der Energie oder dem Wasser sparsam umgehen zu können, sind solche Kurven fast unentbehrlich. Auch Ihnen werden ohne langes Nachdenken weitere Einsatzgebiete einfallen.

Natürlich sind solche Grafiken für den Amiga keine besondere Herausforderung. Mit dem bisher Gelernten können Sie wahrscheinlich diese Grafiken auch ohne Anleitung erstellen. Nun, dieses Kapitel will zwei Fliegen mit einer Klappe erschlagen. Zum ersten, wie bereits erwähnt, will es die verschiedenen grafischen Darstellungsformen zeigen. Da mit diesem Kapitel der erste Teil des Buches beendet wird, stellt es zum zweiten eine Zusammenfassung des bisher Erarbeiteten dar. So werden in den Programmen unter anderem die Anweisungen PATTERN, AREA und AREAFILL verarbeitet. Auch bei der Kreisdarstellung gibt es noch einige Aspekte, die sehr interessant sind.

5.1 Die Eingabe

Bevor Daten grafisch dargestellt werden können, müssen sie natürlich erst vorhanden sein. Eine Möglichkeit ist das Einlesen von Daten aus einer Datei. Die Daten können dann in einem Programm ergänzt, abgespeichert oder einfach auf dem Bildschirm dargestellt werden. Am komfortabelsten ist natürlich die Datenübertragung mittels DFÜ von einer zentralen Datenerfassung. In unseren Beispielen werden wir die Daten ganz banal über die Tastatur eingeben. Dazu schreiben wir ein kleines Programm, das für alle Arten der grafischen Darstellung in diesem Kapitel verwendet wird.

Die Handhabung des Programmes ist sehr einfach. Nach dem Programmstart geben Sie den Namen der Grafik, zum Beispiel »Umsatz«, ein. Nun geben Sie die Bezeichnung und den Wert durch ein Komma getrennt ein:

```
Nürnberg,2ØØØØ
München,4ØØØØ
Dortmund,25ØØØ
Hamburg,1ØØØØ
Kiel,5ØØØ
Ø,Ø
```

Die beiden Nullen, ebenfalls durch ein Komma getrennt, beenden die Eingabe. Die Grafik wird gezeichnet. Anschließend können Sie, so oft Sie wollen, die Fenstergröße verändern. Die Grafik wird jedesmal richtig gezeichnet. Wird das Fenster allzu klein, können natürlich die Texte nicht mehr dargestellt werden. Mit dem Druck auf eine beliebige Taste beenden Sie das Programm.

```
REM Teil 1
'=========
Eingabeprogramm:
IF FRE(-1) <17ØØØØ& THEN PRINT "Speicher zu klein":END
DEFINT a-z :DIM wt(1ØØ),bez-(1ØØ),pat(1),pat1(1)
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
SCREEN 1,64Ø,sh,4,2
WINDOW 2,,,1,1
PALETTE Ø,.6,.6,.6
INPUT "Name der Grafik: ",nam- :PRINT :PRINT:COLOR 3
PRINT "Bitte Wert als kurze Ganzzahl (max.32767) eingeben"
COLOR 1 :PRINT :PRINT
PRINT "Bitte geben Sie nun die Daten ein. Ø,Ø = keine weiteren
Daten"
a=1:n=Ø:max=1
WHILE a
  INPUT "Bezeichnung, Wert: ",bez-(n),wt(n)
  IF bez-(n)="Ø" OR wt(n)<=Ø THEN a=Ø
  IF wt(n)>max THEN max=wt(n)
  GesWert&=GesWert&+wt(n):n=n+1
WEND
n=n-1
ablauf:
wb=WINDOW(2):wh=WINDOW(3)
GOSUB zeichnen
```

```
COLOR 2:LOCATE 1,1Ø:PRINT " Bitte Taste druecken ";
a=-1:b=-1
WHILE a
  IF wb<>WINDOW(2) OR wh<>WINDOW(3) THEN a=Ø
  ta-=INKEY-:IF ta-<>"" THEN b=Ø:a=Ø
WEND
IF b THEN ablauf
WINDOW CLOSE 2: SCREEN CLOSE 1
ERASE wt,bez-,pat,pat1
END

zeichnen:
```

Schauen wir uns kurz das Eingabeprogramm an. Zuerst fragen wir den freien Speicher ab. Da der Screen bereits 82000 Byte Speicher verschlingt und für die PAINT- und AREA-Anweisungen ein Speicher von beträchtlicher (gleicher) Größe benötigt wird, sind die 170000 Byte nicht überzogen. Ist zu wenig Speicher vorhanden, kann es bei den Füll-Befehlen zum Besuch des großen Guru (Dead End Alert) kommen. Anschließend werden die Felder für die Eingaben und die PATTERN-Anweisung dimensioniert. Die Eingaben sind mit 101 Feldelementen ausreichend dimensioniert, da bei größeren Mengen keine ordentliche Grafik mehr gezeichnet werden kann. Nach dem Hires-Bildschirm mit einer Farbtiefe 4 für 16 Farben wird ein Fenster mit einem Größen-Gadget (SizingGadget) geöffnet. Die Hintergrundfarbe wird mit der PALETTE-Anweisung auf ein helles Grau eingestellt. Auf diesem Hintergrund sind die restlichen 15 Farben gut zu erkennen. Es folgen die Eingabeanweisungen.

In der WHILE/WEND-Schleife werden die Daten aufgenommen. Die String-Variable *bez$()* speichert die Benennungen und in das Feld *wt(n)* werden die zugehörigen Werte eingetragen. Da wir zu Programmbeginn mit DEFINT a–z die Variablen als kurze Ganzzahlen definiert haben, führen andere Eingaben zu einer Fehlermeldung. Wollen Sie andere Variablen mit dem Programm verarbeiten, so genügt es nicht, die Definition zu ändern. Auch die Berechnung der Y-Achse und die Ausgabe der Werte sind davon betroffen. Die Ziffer Null bei den Bezeichnungen zeigt an, daß die Eingabe beendet werden soll. Die Schleifen-Variable *a* wird außerdem unwahr, wenn negative Werte eingegeben werden. Diese ließen sich theoretisch auch grafisch darstellen, die einzelnen Subroutinen sind jedoch dafür nicht ausgelegt. In den Variablen *max* wird der größte Eingabewert und in *GesWert&* die Summe aller Eingaben gespeichert. Nachdem alle Werte eingegeben sind, kann zur Subroutine zum Zeichnen der Grafik verzweigt werden. Vorher werden beim Label »ablauf« die aktuellen Fensterabmessungen in den Variablen *wb* und *wh* festgehalten. Nach der Rückkehr vom Zeichnen der Grafik wird in einer weiteren WHILE/WEND-Schleife geprüft, ob das Programm beendet werden soll (beliebiger Tastendruck), oder ob sich die Fensterabmessungen verändert haben. In diesem Fall wird erneut zur Subroutine »zeichnen« verzweigt.

5.2 Das Linien-Diagramm

Bevor wir uns Gedanken darüber machen, wie so eine Grafik am einfachsten zu programmieren ist, werden wir uns überlegen, wie und was eigentlich dargestellt werden soll. Dazu legen wir zuerst fest, daß die Grafik zweidimensional ausgegeben wird. Damit stehen uns die Koordinaten X und Y in einem Koordinatenkreuz zur Verfügung. Eine der Koordinaten nehmen wir für den Wert einer Sache. Aus zeichnerischen Gründen ist die Y-Koordinate dafür geeignet. Auf der X-Koordinaten können wir nun darstellen, was für ein Wert ausgegeben wird. Nehmen Sie zum Beispiel den Energieverbrauch Ihres Fahrzeuges. Die Y-Koordinate gibt dabei den Verbrauch in l/100 km an (oder in Litern, oder als Wert in DM). Auf der X-Achse wird nun eingetragen, wann dieser Verbrauch stattgefunden hat (Januar, Februar etc.). Die Verbindung der einzelnen Werte ergibt eine Kurve oder eine mehr oder weniger gezackte Linie.

Wollen Sie diese Grafik auf verschiedene Fahrzeuge ausweiten, so können Sie eine dritte Koordinate, die Z-Achse hinzufügen. Auf Ihr können Sie dann die Monatswerte der jeweiligen Fahrzeuge abtragen. Damit hätten Sie dann eine dreidimensionale Grafik, auf die wir in einem eigenen Kapitel eingehen werden. Eine andere Möglichkeit bietet die Anlage weiterer Linien, die zur Unterscheidung voneinander in verschiedenen Farben gezeichnet werden. Wir werden der Einfachheit halber vorerst auf eine dritte Ebene, wie sie auch aussehen mag, verzichten.

Bevor wir auf weitere Einzelheiten eingehen, sehen Sie sich erst einmal an, wie eine solche Grafik aussehen kann:

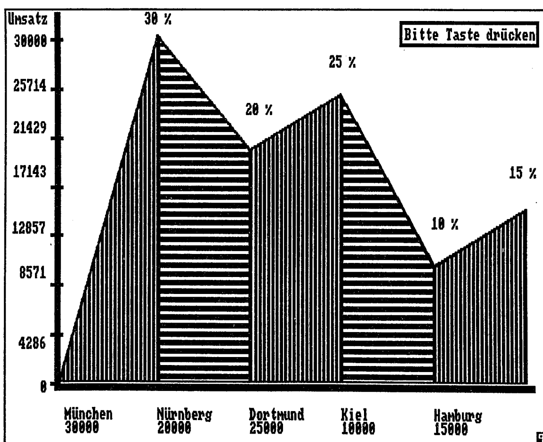


Bild 5.1:
Grafische Darstellung von Zahlen als Linien-Diagramm

Die Skizze sieht doch recht einfach aus. Die Programmierung wird ein Kinderspiel sein. Doch ganz so einfach, wie es aussieht, ist es doch nicht. Wir stellen noch einige Anforderungen an die Grafik. Die erste Forderung lautet, daß die Grafik in jeder beliebigen Fenstergröße gezeichnet werden kann. Oder anders ausgedrückt, bei einer Veränderung der Fenstergröße muß die Grafik ohne Parameterübergabe neu berechnet und sauber gezeichnet werden. Diese Forderung gilt für alle Darstellungen der Geschäftsgrafiken in diesem Kapitel.

Links von der Y-Achse sind die einzelnen Werte neben einer Art von Maßstab eingetragen. Die Größe der Zahl bestimmt daher den benötigten Platz und die Lage der Y-Achse. Der Platzbedarf muß vom Programm ermittelt werden. Die Ausgabe des Prozentsatzes erfolgt über den Linien der Grafik. Die Textausgabe des Satzes benötigt also unterschiedliche Zeilenwerte. Sie sehen an den drei genannten Werten, daß es ohne Rechnerei nicht abgehen wird.

In dem Kapitel über die Eingabe haben Sie erfahren, daß, solange das Programm läuft, jede Fensterveränderung registriert und jedesmal die Grafik neu gezeichnet wird. Damit sind wir bereits wieder bei der Kernforderung, die an das Zeichen-Programm zu stellen ist. Als Bezugsgrößen für alle anderen Maße der Grafik dienen uns die Fensterbreite und die Fensterhöhe. Wir finden diese in den WINDOW-Funktionen zwei und drei und halten sie in den Variablen *xb* und *yh* fest. Die dritte Bezugsgröße für die Lage der Y-Achse holen wir uns aus dem größten Eingabewert *max*. Dazu wandeln wir den Wert in eine Zeichenkette um und ermitteln mit der LEN-Anweisung die Anzahl der Ziffern. Zur Berechnung der Breite der Ziffern in *wlg* addieren wir eine Ziffer hinzu und multiplizieren das Ergebnis mit der Pixelbreite (8) eines Schriftzeichens. Damit können wir auch die Breite einer einzelnen Grafik ermitteln. Dazu reduzieren wir die Fensterbreite *xb* um den Platz für die Ziffern *wlg*, einem Sicherheitszuschlag (20) und teilen das Ergebnis durch die Anzahl der Eingaben *n*. Fehlt uns nur noch die Höhe der Grafik. Dazu legen wir einen Faktor *fakt!* fest, der uns aus dem jeweiligen Wert die Höhe der Grafik berechnen läßt:

```
fakt!=Yrest/max            'Wert*Faktor=Linienhoehe
```

Diesen Faktor erhalten wir, wenn wir den verbleibenden Rest der Y-Koordinate *Yrest* durch den maximalen Wert *max* dividieren.

Ausgehend von diesen fünf Grundwerten können wir nun die Koordinatenwerte berechnen. Sie sind in dem folgenden Programm ausreichend kommentiert:

```
REM Teil 2
'=========
'P5-1
Linien  Pfad: Grundlagen/5GeschGrafik/Linien
'Fuer Fenster von beliebiger Groesse!
```

```
'Werte
xb=WINDOW(2):yh=WINDOW(3)
wl-=STR-(max):wlg=(LEN(wl-)+1)*8 'Pixelbreite des Wertes
Linienbreite!=((xb-wlg-2Ø)/n)    'Breite Linienflaeche
IF Linienbreite! <2 THEN BEEP:RETURN
br=FIX(Linienbreite!)
platz=FIX(br/8)           'max. Laenge der Bezeichnung
Xstart=wlg+12             'Linie min. X
Ystart=yh-35              'Linie max. Y
Yrest=Ystart-15           'maximale Linienhoehe
fakt!=Yrest/max           'Wert*Faktor=Linienhoehe
'Achsenkreuz
CLS:PRINT nam-
LINE (wlg,Ø)-(wlg+6,yh-31),1,bf    'Y-Achse
LINE (wlg+7,yh-34)-(xb,yh-31),1,bf 'X-Achse
IF max <= 7 THEN z=max ELSE z=7    'Unterteilung Y-Achse
zl!=Yrest/z :maxTeil!=max/z
FOR i=Ø TO z
  yps=Ystart-i*zl!:zlpos=yps/8
  IF zlpos<1 THEN zlpos=1
  LOCATE zlpos,1:PRINT CINT(maxTeil!*i)
  LINE (wlg-5,yps)-(wlg+9,yps+1),2,bf
NEXT
'Zeichnen
farbe=2:YAlt=Ystart
FOR i = Ø TO n-1
  farbe=farbe+1:IF farbe>15 THEN farbe=3
  COLOR farbe
  Xpos=Xstart+i*br :Ymax=Ystart-(wt(i)*fakt!)
  proz=wt(i)/(GesWert&/1ØØ)          'Prozentsatz
  AREA (Xpos,Ystart)
  AREA (Xpos+br,Ystart)
  AREA (Xpos+br,Ymax)
  AREA (Xpos,YAlt)
  AREA (Xpos,Ystart)
  AREAFILL
  text-=bez-(i):textLg=LEN(text-)    'Bezeichnung des Wertes
  IF textLg>platz THEN               'max. Laenge der Bezeichnung
    textLg=platz:text-=LEFT-(text-,textLg)
  END IF
```

```
  Yunt=(Ystart+2Ø)/8
  Yoben=Ymax/8:IF YAlt/8<Yoben THEN Yoben =YAlt/8
  IF Yoben<1 THEN Yoben=1
  LOCATE Yunt:PRINT PTAB(Xpos):PRINT text-;  'Bezeichnung schr.
  LOCATE Yunt+1:PRINT PTAB(Xpos-8):PRINT wt(i); 'Wert schreiben
  LOCATE Yoben:PRINT PTAB(Xpos-3):PRINT proz "%"; '%-Satz schr.
  YAlt=Ymax
  NEXT
RETURN
```

Über die Ermittlung der Ausgangswerte haben wir bereits vor Programmbeginn gesprochen. Fangen wir also nach dem Zeichnen des Achsenkreuzes an. In der WHILE/WEND-Schleife werden maximal 8 Unterteilungen auf der Y-Achse gezeichnet und die zugehörigen Werte errechnet und dazugeschrieben. Das Linien-Diagramm wird in der FOR/NEXT-Schleife gezeichnet. Um die Grafik optisch aufzuwerten, zeichnen wir nicht nur die Verbindungslinien zu den einzelnen Werten, sondern ein ausgemaltes Viereck. Die Farben werden jeweils nacheinander mit den Nummern 3 bis 15 gesetzt. Bevor das Polygon gezeichnet wird, errechnen wir für die Grafik die Startposition auf der X-Achse *Xpos*, die maximale Höhe in der Y-Achse *Ymax* und den Prozentsatz *proz* vom Gesamtwert.

Die Grafik wird mit AREA-Anweisungen gezeichnet und mit AREAFILL ausgemalt. Anschließend wird überprüft, ob die Bezeichnung zu lang ist. In diesem Fall wird der Text am Ende abgeschnitten. Ist dies getan, so können die Y-Positionen, oder besser gesagt die Textzeilen für die drei Ausgaben: Bezeichnung *text$*, Wert *wt()* und Prozentsatz *proz*, errechnet werden. Nachdem die Texte ausgegeben sind, kann die nächste Teilgrafik gezeichnet werden, bis die Anzahl *n* erreicht ist.

5.3 Das Balken-Diagramm

Beim Balken-Diagramm verfahren wir ähnlich wie beim Linien-Diagramm. Der einzige Unterschied liegt darin, daß die einzelnen Werte nicht durch Linien verbunden, sondern durch einen Balken dargestellt werden. Wir haben es also wieder mit zwei Ausgaben in der X- und Y-Achse zu tun. Soll eine dritte Ausgabe hinzugefügt werden, so bietet sich wiederum die Z-Achse an, auf der die einzelnen Balken der X-Y-Ebene hintereinander gezeichnet werden. Eine andere Variante ist die, daß verschiedenfarbige Balken nebeneinander gezeichnet werden. Nehmen wir an, auf der Y-Achse ist der Preis pro kg eingetragen, die X-Achse zeigt vergleichbare Maschinenbauteile, und nun liegen verschiedene Angebotspreise vor. Dann versieht man jeden Anbieter mit einer bestimmten Farbe und stellt die Balken der Anbieter für jedes einzelne Maschinenbauteil nebeneinander. Damit kann man auf einen Blick das Preisgefüge überblicken. Wir werden in unserem Beispiel wieder bei der zweidimensionalen Darstellung bleiben.

Da ein Diagramm mit den nebeneinander gestellten Balken optisch keinen besonderen Eindruck macht, werten wir die Grafik etwas auf. Dazu wandeln wir die Balken in Quader um. Als Blickrichtung legen wir rechts oben fest. Die rechte Ansicht und das obere Viereck sollen, zur Darstellung einer Schattenwirkung, farblich abgehoben werden. Wie können wir das am besten bewerkstelligen? Die eine Möglichkeit ist eine farbliche Abstimmung nebeneinander liegender Farben. Bei nur 16 Farben abzüglich den Farben für das Achsenkreuz und die Beschriftung bleiben uns dann für die Balken nur 6 bis 7 Farbunterscheidungen. Da überlegen wir uns lieber, was der Amiga sonst noch anbieten kann. Es gibt ja noch die PATTERN-Anweisung, mit der Flächen mit einem bestimmten Muster gefüllt werden können. Als Fläche wählen wir ein einfaches Punkt-leer-Punkt-Muster. Damit läßt sich gut eine Schattenbildung realisieren. Wie bereits eingangs erwähnt, ist das folgende Programm nicht vollständig. Um es lauffähig zu machen, müssen Sie das Programm Teil 1 davorsetzen:

```
'===================
REM Teil 1 einfuegen
'===================

REM Teil 3
'=========
'P5-2
Balken  Pfad: Grundlagen/5GeschGrafik/Balken
'Fuer Fenster von beliebiger Groesse!
'Werte
xb=WINDOW(2):yh=WINDOW(3)
wl-=STR-(max):wlg=(LEN(wl-)+1)*8 'Pixelbreite des Wertes
Balkenbreite!=((xb-wlg-2Ø)/n)/2  'Breite des Balkens
IF Balkenbreite! <2 THEN BEEP:RETURN
br=FIX(Balkenbreite!)
platz=FIX(2*br/8)          'max. Laenge der Bezeichnung
Xstart=wlg+12              'Balken min. X
Ystart=yh-35               'Balken max. Y
Yrest=Ystart-(br/2)-15     'maximale Balkenhoehe
Abstand=(xb-wlg-12)/n      'Abstand der Balken
fakt!=Yrest/max            'Wert*Faktor=Balkenhoehe
'Achsenkreuz
CLS:PRINT nam-
LINE (wlg,Ø)-(wlg+6,yh-31),1,bf    'Y-Achse
LINE (wlg+7,yh-34)-(xb,yh-31),1,bf 'X-Achse
IF max <= 7 THEN z=max ELSE z=7    'Unterteilung Y-Achse
zl!=Yrest/z :maxTeil!=max/z
FOR i=Ø TO z
```

```
  yps=Ystart-i*zl!:zlpos=yps/8:IF zlpos<1 THEN zlpos=1
  LOCATE zlpos,1:PRINT CINT(maxTeil!*i)
  LINE (wlg-5,yps)-(wlg+9,yps+1),2,bf
NEXT
'Zeichnen
farbe=2
pat(Ø)=&HAAAA:pat(1)=&H9999:pat1(Ø)=&HFFFF:pat1(1)=&HFFFF
FOR i = Ø TO n-1
  farbe=farbe+1:IF farbe>15 THEN farbe=3
  COLOR farbe
  Xpos=Xstart+i*Abstand :Ymax=Ystart-(wt(i)*fakt!)
  proz=wt(i)/(GesWert&/1ØØ)          'Prozentsatz
  LINE (Xpos,Ystart)-(Xpos+br,Ymax),farbe,bf   'Balkenrechteck
  'Perspektive
  PATTERN ,pat
  LINE (Xpos+br,Ystart)-(Xpos+br+br/2,Ystart-br/2),2
  LINE -(Xpos+br+br/2,Ymax-br/2),2
  LINE -(Xpos+br/2,Ymax-br/2),2
  LINE -(Xpos,Ymax),2
  LINE -(Xpos+br,Ymax),2
  LINE -(Xpos+br,Ystart),2
  LINE (Xpos+br+br/2,Ymax-br/2)-(Xpos+br+2,Ymax-2),2
  PAINT (Xpos+br+1,Ystart-2),farbe,2
  PATTERN ,pat1
  text–=bez–(i):textLg=LEN(text–)       'Bezeichnung des Wertes
  IF textLg>platz THEN              'max. Laenge der Bezeichnung
    textLg=platz:text–=LEFT–(text–,textLg)
  END IF
  Yunt=(Ystart+2Ø)/8:Yoben=(Ymax-br/2)/8
  IF Yoben<2 THEN Yoben=2
  LOCATE Yunt:PRINT PTAB(Xpos):PRINT text–;'Bezeichnung schreiben
  LOCATE Yunt+1:PRINT PTAB(Xpos-8):PRINT wt(i); 'Wert schreiben
  LOCATE Yoben-1:PRINT PTAB(Xpos-3):PRINT proz "%";'%-Satz schr.
NEXT
RETURN
```

Einschließlich dem Liniensetzen der Werte-Skala ist das Programm ähnlich aufgebaut wie das vorhergehende Programm. Die Berechnung der einzelnen Variablen aus der Fensterhöhe und Fensterbreite ist ausreichend kommentiert. Bei der Markierung »Zeichnen« setzen wir die Muster für die PATTERN-Anweisung. Das erste Feld *pat()* zeichnet das besprochene Muster und das zweite Feld *pat1()* stellt den normalen Zustand wieder her.

Für das Prisma wird erst das Rechteck mit einer LINE-Anweisung und der Option *bf* gezeichnet. Für die perspektivische Darstellung wird ein Polygon aus lauter LINE-Anweisungen konstruiert. Dabei wird die Linie von der rechten äußeren Ecke bis zur rechten oberen Ecke des Rechteckes nicht ganz durchgezogen. Dadurch läuft die Farbe bei der folgenden PAINT-Anweisung auch in das Viereck der Draufsicht. Da mit PATTERN vorher das Muster gesetzt wurde, wird der perspektivische Bereich des Prismas mit einem Punktmuster der aktuellen Farbe gefüllt. Nachdem mit der zweiten PATTERN-Anweisung der normale Zustand wiederhergestellt wurde, können die Texte geschrieben werden. Nach *n* Durchgängen ist die Grafik fertig gezeichnet.

5.4 Die Kuchen-Grafik

Die optisch eindrucksvollste unter den Geschäftsgrafiken ist zweifellos die Kuchengrafik. Allerdings ist sie für eine dreidimensionale Darstellung kaum geeignet. Man könnte höchstens die Kuchenscheiben stark elliptisch zeichnen und mehrere Scheiben übereinander ausgeben, um eine dritte Dimension zu gewinnen. Auf eine Prinzip-Skizze können wir sicherlich verzichten.

Um die Kuchengrafik optisch noch besser zu gestalten, kann man einen Kreisausschnitt, den man besonders hervorheben will, etwas nach außen zeichnen. Außerdem sieht man auch oft eine perspektivische Darstellung, in der für die Torte eine Höhe eingezeichnet wird. Selten finden Sie allerdings die Textausgabe wie in dem gezeigten Beispiel. Dazu ist etwas Rechnerei erforderlich, und um diese zu vermeiden, finden Sie meistens die Werte in Tabellenform neben die Grafik geschrieben.

Wie konstruieren wir am besten die Grafik? Dazu benötigen wir zuerst eine Variable, die uns sagt, für welchen Wert ein Grad des Kreisausschnittes gezeichnet werden soll:

```
Faktor!=360/GesWert&
```

Diesen Faktor erhalten wir, wenn wir die 360 Grad eines vollen Kreises durch den Gesamtwert *GesWert&* dividieren. Mit diesem Faktor multiplizieren wir den Einzelwert *wt()* und erhalten den Winkel des Kuchenstückes. Da der Amiga einen Winkel im Bogenmaß erwartet, rechnen wir diesen um *(wk!)*. Um den Platz des Kreisausschnittes im Kreis festzulegen, addieren wir diesen zum Gesamtwinkel. Den Endwinkel erhalten wir dann logischerweise, wenn wir zum Startwinkel den Winkel des Kuchenstückes addieren. Im Zusammenhang sieht das folgendermaßen aus:

```
w!=wt(i)*Faktor!              'Winkel Kuchenstueck in Grad
wk!=w!*pi!/180                'Winkel Kuchenstueck im Bogenmass
w1!=w2!+.001                  'Startwinkel im Bogenmass
w2!=w2!+wk!                   'Endwinkel im Bogenmass
```

```
FOR j! =.785 TO 6.28 STEP .785  '45 Grad Sicherung
  IF w2!>j!-.ØØ5 AND w2!<j!+.ØØ5 THEN w2!=w2!+.Ø2
NEXT
```

Der Anfang der Grafik, also der Winkel 0 Grad, befindet sich in einer Linie vom Mittelpunkt nach rechts gesehen. Im Bild ist der Anfang mit *Start* gekennzeichnet. Am Ende des Programmausschnittes finden Sie eine kleine Schleife, die bei bestimmten Werten des Endwinkels diesen erhöht. Zu was soll den das gut sein? Dazu muß ich etwas ausholen. In einem der ersten Kapitel haben Sie gelernt, wie ein Kreisausschnitt im Amiga-Basic programmiert wird:

```
CIRCLE (x1,y1),r,2,-w1!,-w2!,verh!
```

Indem die Start- und Endwinkel negativ in die Anweisung eingetragen werden, erhalten wir einen Kreisausschnitt. Eine prima Sache und so spielend einfach zu programmieren! Doch leider hat die Anweisung einen kleinen Schönheitsfehler. Bei Winkeln eines vielfachen von 45 Grad (45, 90, 135 etc.) wird die Linie am Ende des Kreisausschnittes nicht gezeichnet. Beim Füllen des Kuchenstückes mit PAINT läuft Ihnen das ganze Bild voll. Um einem solchen Mißgeschick vorzubeugen, wurde die Schleife programmiert. Sie läßt diese Winkel einfach nicht zu, indem die Werte einfach nach oben gemogelt werden.

Jetzt wissen wir, wie die Tortenstücke gezeichnet werden, und brauchen sie nur noch auszumalen. Dazu benötigen wir aber Koordinatenwerte, die tunlichst innerhalb des Kreisausschnittes liegen sollten. Den Füllpunkt legen wir in die Mitte des Kuchenstückes *wMitte!*, 5 Pixel vom Kreisbogen nach innen. Jetzt müssen uns die Winkelfunktionen helfen. Sollten Sie damit etwas Probleme haben, so finden Sie darüber in einem eigenen Kapitel eine kurze Zusammenfassung. Der ganze Füllvorgang im Zusammenhang könnte dann wie folgt aussehen:

```
wMitte!=wges!+(w!/2)             'Winkel bis Kuchenstueck Mitte
wges!=wges!+w!                   'Winkel vom Start an
faPosX=x1+((r-5)*COS(wMitte!*pi!/18Ø))        'Beginn fuellen X
faPosY=y1-((r-5)*SIN(wMitte!*pi!/18Ø)*verh!)  'dto. y
PAINT (faPosX,faPosY),farbe,2
```

Damit aber noch nicht genug. Es fehlt uns noch die Position für die richtige Positionierung des Textes am äußeren Rand des Kreisausschnittes. Der Mittenwinkel *wMitte!* wird dafür verwendet. Die Positionierung bei diesem Winkel erfolgt in einem Abstand von 10 Pixel außerhalb des Kreisbogens. Das gleiche Problem hatten wir gerade mit den SIN- und COS-Funktionen gelöst. Auch bei der Textpositionierung gilt diese Lösung, aber nur auf der rechten Hälfte des Kreises. Bei der linken Hälfte würden wir ja dann in den Kreis schreiben und den schönen Anblick der Torte vers... Der Text muß daher in diesem Bereich um die Länge des Textes *textLg* nach links verschoben werden.

```
textPosX=x1+((r+1Ø)*COS(wMitte!*pi!/18Ø))
IF wMitte!> 9Ø AND wMitte! <29Ø THEN
  textPosX=textPosX-(8*textLg)
END IF
textPosY=y1-((r+1Ø)*SIN(wMitte!*pi!/18Ø)*verh!)
textPosY=textPosY/8
LOCATE textPosY: PRINT PTAB(textPosX) :PRINT text-;
```

Dazwischen gilt es noch einige Grenzen abzusichern. Diese finden Sie in dem Programmbeispiel.

Daß folgendes Programm alleine nicht lauffähig ist, wissen Sie inzwischen. Setzen Sie deshalb bitte das Programm Teil 1 davor:

```
'===================
REM Teil 1 einfuegen
'===================
'
REM Teil 4
'=========
'P5-3
'Fuer Fenster von beliebiger Groesse
Kuchen  Pfad: Grundlagen/5GeschGrafik/Kuchen
x1=WINDOW(2)/2:y1=WINDOW(3)/2
pi!=3.141593 :verh!=.52
bb=x1/5*3 :hh=y1*3/2
IF bb<hh THEN
  r=bb: platz=(x1-r)/8
ELSE
  r=hh:platz=(x1-r)/8
END IF'
Faktor!=36Ø/GesWert&
CLS
PRINT nam-
farbe=2:w2!=Ø:wges!=Ø
FOR i = Ø TO n-1
  farbe=farbe+1:IF farbe>15 THEN farbe=3
  COLOR farbe
  w!=wt(i)*Faktor!  'Winkel Kuchenstueck in Grad
  wk!=w!*pi!/18Ø    'Winkel Kuchenstueck im Bogenmass
  wMitte!=wges!+(w!/2)  'Winkel bis Kuchenstueck Mitte
  wges!=wges!+w!    'Winkel vom Start an
```

```
  w1!=w2!+.ØØ1        'Startwinkel im Bogenmass
  w2!=w2!+wk!         'Endwinkel im Bogenmass
  FOR j! =.785 TO 6.28 STEP .785  '45 Grad Sicherung
    IF w2!>j!-.ØØ5 AND w2!<j!+.ØØ5 THEN w2!=w2!+.Ø2
  NEXT
  proz=wt(i)/(GesWert&/1ØØ)          'Prozentsatz
  IF w2!>6.28 THEN w2!=6.28    'Obergrenze sichern
  CIRCLE (x1,y1),r,2,-w1!,-w2!,verh!
  faPosX=x1+((r-5)*COS(wMitte!*pi!/18Ø))  'Beginn fuellen X
  faPosY=y1-((r-5)*SIN(wMitte!*pi!/18Ø)*verh!)  'dto. y
  PAINT (faPosX,faPosY),farbe,2
  text-=STR-(wt(i))+" "+bez-(i)+STR-(proz)+"%"
  textLg=LEN(text-)
  IF textLg>platz THEN            'max. Laenge der Bezeichnung
    textLg=platz:text-=LEFT-(text-,textLg)
  END IF
  textPosX=x1+((r+1Ø)*COS(wMitte!*pi!/18Ø))
  IF wMitte!> 9Ø AND wMitte! <29Ø THEN
    textPosX=textPosX-(8*textLg)
  END IF
  IF textPosX<Ø THEN textPosX=Ø
  textPosY=y1-((r+1Ø)*SIN(wMitte!*pi!/18Ø)*verh!)
  textPosY=textPosY/8:IF textPosY<1 THEN textPosY=1
  IF wMitte!> 22Ø THEN:textPosY=textPosY+1
  LOCATE textPosY: PRINT PTAB(textPosX)
  PRINT text-;
NEXT
RETURN
```

Um die Grafik unabhängig von der Fenstergröße zeichnen zu können, benötigen wir wieder die Fensterbreite und Fensterhöhe aus den WINDOW-Funktionen. Den Mittelpunkt des Kreises legen wir damit exakt in die Mitte des Fensters. Da wir in einem Screen von hoher Auflösung arbeiten, müssen die Y-Werte des Kreises ins richtige Verhältnis gebracht werden (*verh!=.52*). In der folgenden IF-Anweisung wird der Radius *r* abhängig vom Verhältnis Fensterbreite zu Fensterhöhe berechnet und der zur Verfügung stehende Platz *platz* für die Beschriftung ermittelt.

Die Berechnung der Winkel haben wir bereits besprochen. Da sich bei der Addition der einzelnen Winkel eine kleine Ungenauigkeit einschleichen kann, müssen wir die Obergrenze absichern. Der maximale Winkel im Bogenmaß beträgt 2*Pi. Wird dieser Wert überschritten, meldet sich der Basic-Interpreter mit einer Fehlermeldung. Nach dem Zeichnen und Füllen des Kuchenstückes wird der Text für den Kreissektor zusammen-

gestellt. Zuerst kommt der Wert *wt()* gefolgt von einem Leerzeichen. Es schließt sich die Bezeichnung *bez$()* und der prozentuale Anteil *proz* mit einem Prozentzeichen an:

```
text-=STR-(wt(i))+" "+bez-(i)+STR-(proz)+"%"
```

Da dieser lange Text nicht immer Platz neben der Grafik findet, wird die Länge mit dem zur Verfügung stehenden Platz verglichen und, wenn notwendig, gekürzt. Die Text-Positionen werden noch auf eine eventuelle Unterschreitung der erlaubten Grenzen (außerhalb des Fensters) geprüft und gegebenenfalls korrigiert. Damit kann endlich die Beschriftung des Kreisausschnittes ausgegeben werden.

Bei der grafischen Darstellung der Tortengrafik hatten wir etwas mehr rechnen müssen als bei den anderen Grafiken. Das Ergebnis zeigt uns jedoch, daß sich der Aufwand gelohnt hat.

Grafik mit AMIGA-BASIC

Darstellung

2. Teil

Commodore Sachbuch

Kapitel 6

Die Routinen des Betriebssystems

Daß der Amiga, mit seiner fortschrittlichen Technologie, immens viel zu bieten hat, das wissen Sie längst. Bereits mit seinen normalen Basic-Befehlen bietet er eine Fülle von Grafik-Fähigkeiten, von den andere Computer-Besitzer nur träumen können. Einen Teil der Grafik-Möglichkeiten des Standard-Basic haben Sie im ersten Teil des Buches bereits zu sehen bekommen. Im weiteren Verlauf des Buches werden Sie noch einiges mehr davon kennenlernen.

Aber der Amiga kann mehr, als sein Basic zu bieten hat. Ganz richtig ist diese Aussage allerdings nicht. Das Amiga-Basic hat in seinem normalen Befehls-Vorrat einige Funktionen und Anweisungen, die das Basic mächtig erweitern können. Wir gelangen damit in andere Dimensionen der Programmierung. In Dimensionen, die eigene Gesetzmäßigkeiten haben, die beim Programmieren viel Vorsicht verlangen, die als Entschädigung dafür aber fast alles das bieten, für das Sie sonst eigene Programmiersprachen erlernen müßten.

Den wenigen Worten haben Sie sicherlich entnommen, daß es hier um ein Thema geht, das sich nicht in einigen Sätzen abhandeln läßt. Sie finden in diesem Buch jedoch alles das, was Sie zur Programmierung der Library-Routinen (Erklärung folgt gleich) und Techniken für die Grafik-Programmierung benötigen. Allerdings sind die Erklärungen sehr kurz gefaßt. Der Grund dafür ist sehr einfach. Eine detaillierte Auseinandersetzung mit diesem Thema füllt ein eigenes Buch und ein reines Grafik-Buch ist dafür nicht geeignet. Wollen Sie Ihr Wissen zu diesem Thema vertiefen und wollen Sie weitere Möglichkeiten der Betriebssystem-Routinen kennenlernen, so empfehle ich Ihnen mein Buch »Programmierpraxis Amiga-Basic« aus dem gleichen Verlag. Sie finden darin auch etliches aus dem großen Bereich der Grafik, das in diesem Buch nur gestreift wird. Dazu zählen die Schriftdarstellung in sämtlichen Schriftarten, die Screen- und Window-Manipulationen, Gadgets, Requester, Spezial-Menüs etc.

An dieser Stelle muß ich auch eine Warnung anbringen. Bei der Programmierung mit den Routinen der Libraries ist besondere Vorsicht geboten. Es handelt sich dabei in der Regel um Maschinenprogramme. Wenn Sie jedoch ein Maschinenprogramm aufrufen,

so ist die Kontrolle voll auf das Maschinenprogramm übergegangen. Während dieser Zeit bemerkt der Basic-Interpreter keinen Fehler und kann Ihnen also auch keinen Fehler melden. Er kann Sie deshalb auch nicht beschützen, indem er Fehler abfängt und das Programm unterbricht. Sie müssen eventuelle Fehlerquellen selbst ausschalten. Die einzigen Meldungen, die Sie erhalten, kommen direkt vom Betriebssystem. Diese heißen dann oft »Dead-End-Alert«. Manchmal, bei besonders schweren Fehlern, kann der Amiga nicht einmal mehr diese ausgeben. Dann gibt es nur noch eines, Disketten raus (wenn das Licht am Laufwerk erloschen ist) und neu starten. Ich will Sie damit beileibe nicht erschrecken. Viele der Routinen sind harmlos und verzeihen (ignorieren) Fehler, aber andere ... Nun, alle Programme dieses Buches laufen fehlerfrei. Wenn Sie sich an den beschriebenen Weg halten und die Variablen richtig definieren, kann auch bei eigenen Programmen kaum etwas passieren. Auf besondere Gefahren, die mir bekannt sind, werden Sie jeweils in den Beschreibungen der einzelnen Programme hingewiesen. Nun denn frisch drauf los! Dem GURU werden wir die Zähne zeigen.

6.1 Die Libraries

Die Software des Amiga besteht aus einer Anzahl von leistungsfähigen Software-Modulen. Ein Teil dieser Module befindet sich ständig in dem geschützten Kickstart-Bereich. Das ist der Bereich, der bei den ersten Amigas von der Kickstart-Diskette in den WOM-Bereich geladen wurde. Bei den heutigen Amigas ist er in einem ROM-Baustein untergebracht. Außerdem sind noch Programm-Module auf der Workbench-Diskette vorhanden, die bei Bedarf nachgeladen werden. Ein Teil dieser System-Module sind die Libraries, also Bibliotheken. Jede dieser Bibliotheken besteht aus einer Anzahl von wirkungsvollen Maschinenprogrammen. Durch diese offene Struktur können einzelne Routinen, ja sogar ganze Libraries hinzugefügt oder geändert werden. Auf diese Routinen des Amiga-Betriebssystems können wir auch vom Basic aus zugreifen. Für die Grafik-Programmierung werden wir hauptsächlich Routinen aus folgenden Libraries einsetzen:

`dos.lib`

Diese Bibliothek ist für die Befehle des oder der Diskettenlaufwerke zuständig.

`exec.lib`

Exec bildet den Kern der Software unseres Amigas. Dieser Manager der System-Software verteilt die Prozessorzeit, den Speicherplatz, verwaltet die Tasks und die anderen Bibliotheken.

`graphics.lib`

Diese Library werden wir weidlich nutzen. Sie stellt die Routinen der untersten Grafikebene zur Verfügung,

`intuition.lib`

Ohne Intuition gäbe es keine Fensterverwaltung, Mausbedienung und Menüs. Auch von dieser Library werden wir regen Gebrauch machen.

6.2 Die .bmap-Dateien

Jetzt haben Sie ein wenig über die Libraries gelesen. Da interessiert es Sie sicherlich auch, wie Sie an diese gelangen können. Zuerst brauchen Sie eine Zugriffsmöglichkeit zu den Bibliotheken. Auf der Diskette *ExtrasD*, die Sie mit Ihrem Amiga erhalten haben, finden Sie eine Schublade *FD1.2*. In der Schublade befindet sich ein Verzeichnis mit dem Namen Basic-FD-Dateien. Wenn Sie Lust haben, können Sie den Inhalt dieses Directorys vom CLI aus lesen. Sie finden dort eine Reihe von .fd-Dateien abgelegt. Es handelt sich dabei um besondere Informations-Dateien, in welchen die erforderlichen Parameter und die verwendeten Register des Prozessors aufgeführt sind (Beispiel: graphics_lib.fd). Mit diesen Dateien kann das Amiga-Basic herzlich wenig anfangen.

Was muß denn dann getan werden, damit das Basic mit den Dateien arbeiten kann? Dazu müssen Sie die .fd-Dateien in .bmap-Dateien konvertieren. In der Schublade »BasicDemos«, auf der gleichen Diskette, finden Sie ein Programm mit dem Namen »ConvertFD«. Mit diesem Programm, das ausführlich kommentiert ist, können Sie die Umwandlung vornehmen. Dabei sollten Sie noch eine Kleinigkeit beachten. Auf der Diskette zu diesem Buch und in sämtlichen Programmen ist ein fester Pfad für die Dateien vorgegeben. Die .bmap-Dateien sind alle in der Schublade »bue« abgelegt. Das Konvertierungsprogramm würde aber die Dateien in das Verzeichnis »libs« der Workbench ablegen. Diese an und für sich logische Ablage bringt aber bei der Arbeit mit den Libraries einen unzumutbaren Aufwand. Laufend müßten Sie während eines Programmablaufes die Workbenchdiskette einlegen.

Für die Programme aus diesem Buch brauchen Sie den genannten Aufwand nicht treiben. Sämtliche benötigten .bmap-Dateien (und noch einige mehr) finden Sie bereits alle konvertiert vor. Wenn Sie jedoch weitere Konvertierungen vornehmen wollen, beachten Sie bitte den oben erwähnten Pfad.

6.3 Basic-Befehle als Library-Schlüssel

Bevor Sie das Maschinenprogramm einer Library-Routine starten können, muß die entsprechende Bibliothek mit der LIBRARY-Anweisung geöffnet werden.

Format: `LIBRARY "Dateiname"`
Beispiel: `LIBRARY "graphics.library"`

Als Dateiname tragen Sie die .bmap-Datei mit Ihrem kompletten Pfadnamen ein. Ohne Pfad kommen Sie aus, wenn Sie vorher mit CHDIR den Weg für Ihr Verzeichnis frei gemacht haben. Mit dieser Vorbereitung können Sie bereits mit CALL eine Library-Routine aufrufen.

Format: `CALL num Variable (Argumentliste)`
Beispiel: `CALL Move&(WINDOW(8),25,3Ø)`

Als *Argumentliste* können Parameter übergeben werden. Diese Liste hängt davon ab, welche Routine aufgerufen wird. Durch den Aufruf der Bibliotheks-Routine »Move« wird eine *num Variable* erzeugt und die Information darüber, wo die Routine im Speicher zu finden ist, der Variablen *Move* zugewiesen. Als Variablen-Typ haben wir eine lange Ganzzahl (Langwort) gewählt, damit die Speicheradresse des Maschinenprogrammes aufgenommen werden kann. Vergessen Sie bitte niemals, den Variablen den richtigen Typ zuzuweisen. Jetzt wissen Sie bereits, wie eine Routine aufgerufen wird. Aber was machen Sie, wenn Sie von dem Maschinenspracheprogramm einen Wert zurückerhalten wollen? Dafür gibt es eine Funktion des Amiga-Basic:

Format: `DECLARE FUNCTION Name Parameterliste LIBRARY`
Beispiel: `DECLARE FUNCTION WritePixel&() LIBRARY`

Wie Sie an dem Beispiel sehen können, muß die *Parameterliste* nicht ausgefüllt werden, da das Amiga-Basic diese Liste ignoriert. Als *Name* wird der Name der Library-Routine eingesetzt. Diese Funktions-Deklaration müssen Sie gesetzt haben, bevor Sie die entsprechende Funktion aufrufen. Am besten eignet sich dafür der Anfang des Programmes. Vermeiden Sie auch, eine Funktion mehrmals zu deklarieren, da das unweigerlich zu einer Fehlermeldung mit Programmabbruch führt. Damit haben wir fast das Repertoire an Basic-Befehlen, die wir für die Programmierung der Library-Routinen benötigen, erschöpft. Bleibt uns nur noch eine Anweisung, die am Ende eines Programmes die Ordnung wiederherstellt:

Format: `LIBRARY CLOSE`

Mit dieser Anweisung werden alle geöffneten Bibliotheken geschlossen.

6.4 Strukturen, Nachschlagewerke der Software

Der Amiga hat eine weiche System-Software. Im Gegensatz zu anderen Computern bedeutet das, daß die Maschinensprache-Routinen des Betriebssystems nicht in festgelegten Speicherstellen liegen. Der Amiga findet seine Informationen in Files, die für die Programmiersprache C und Assembler-Programme angelegt sind. Innerhalb dieser Files sind bestimmte, zusammengehörende Informationen in Daten-Listen, sogenannten Strukturen, zusammengefaßt. Als Beispiel dafür, wie eine Struktur aufgebaut ist, finden Sie anschließend den Anfang der Window-Struktur:

Struktur Window (win& = WINDOW(7))			
Typ	**Bezeichnung**	**Offset**	**Erläuterungen**
Zeiger	NextWindow	win&	Zeiger auf das nächste Fenster
Wort	LeftEdge	+4	linke Kante des Fensters
Wort	TopEdge	+6	obere Kante des Fensters
Wort	Width	+8	Breite des Fensters
Wort	Height	+10	Höhe des Fensters
...			
Langwort	Flags	+24	Fenster-Flags
Zeiger	MenuStrip	+28	Zeiger auf Menü
...			
Byte	PtrHeight	+78	Sprite-Höhe
Byte	PtrWidth	+79	Sprite-Breite
etc.			

An diesem Auszug aus einer Struktur können Sie sehen, welche Art von Daten hinterlegt sind. Da gibt es Zeiger, die auf eine Speicherstelle zeigen, in welcher eine andere Struktur abgelegt ist. Andere Datenfelder enthalten Variablen. In dem Beispiel sind das unter anderem die Variablen für die Lage des Fensters und für die Fenstermaße. Außerdem gibt es noch Variablen für Flags, die für die vielfältigsten Aufgaben benötigt werden. Um eine Speicherstelle lesen zu können, brauchen Sie nur zur Speicherstelle der Struktur den Offset addieren. So erhalten Sie mit

```
PRINT PEEKW(WINDOW(7)+8)
```

die Breite des aktuellen Fensters. Sie haben sicherlich die Tragweite der kurzen Ausführung erkannt. Richtig! Auch beim Amiga können Sie sich alle Informationen holen, die Sie für die Programmierung benötigen. Die Position des Mauszeigers, die Farben, die Screen- und Window-Abmessungen, die Lage der Bitmap und vieles mehr finden Sie in den verschiedenen Strukturen. Bei der Arbeit mit den Strukturen ist es wichtig zu wissen, wie lang die einzelnen Datenfelder sind. Für die Basic-Programmierung können wir die für die Sprache C gültigen Datentypen auf folgende Typen reduzieren:

Type	**Inhalt**	**Länge**	**Bezeichnung in diesem Buch**
Byte	0 bis 255	1 Byte	Byte
Word	0 bis 65535	2 Byte	Wort
Long	0 bis 4294967295	4 Byte	Langwort
Ptr.	Speicheradresse	4 Byte	Zeiger

Da alle Daten positive Werte darstellen, kann es Probleme mit der Zahlendarstellung des Basic geben. Dort haben wir es mit vorzeichenbehafteten Zahlen zu tun. Bei einem Byte und einem Adreßzeiger können keine negativen Werte auftreten. Aufpassen müssen wir jedoch bei einem Wort. Damit der Wert aus dem Datenfeld nicht falsch weiterverarbeitet wird, muß er logisch mit AND 65535 verknüpft werden. Hierzu ein kleines Beispiel. Wollen Sie den aktuellen ViewPort-Modus wissen, so erhalten Sie mit

```
PRINT PEEKW(PEEKL(WINDOW(7)+46)+76)
```

den falschen Wert >>–16384<<. Die Verknüpfung mit AND 65535 dagegen

```
PRINT PEEKW(PEEKL(WINDOW(7)+46)+76)AND 65535
```

liefert das richtige Resultat >>49152<<. Der voreingestellte Wert des Amiga-Basic ist HIRES ($8000) + SPRITES ($4000) = 49152.

Die für die Grafik-Programmierung wichtigen Strukturen sind im Anhang zu finden. Aus den erwähnten Gründen werden sie im Verlauf des Buches nicht separat erklärt. Innerhalb der Programme finden Sie aber reichlich Gelegenheit, den Aufbau der Strukturen nachzuvollziehen.

Das Lesen und/oder Weiterverarbeiten von Daten aus den Strukturen ist nur ein Teilaspekt. Bei der Programmierung der für die Grafik wichtigen Library-Funktionen müssenauch neue Strukturen erstellt werden. Das hört sich dramatischer an als es ist. Dazu reservieren Sie einen Speicherbereich, der groß genug ist, die Daten der Struktur aufzunehmen, und poken die erforderlichen Daten einfach in die einzelnen Speicherstellen. Wie solche Speicherbereiche problemlos programmiert werden können, finden Sie im nächsten Kapitel.

6.5 Benötigten Speicherplatz vergibt das System

Die einfachste Belegung eines Speicherbereiches kennen Sie bereits. Durch die Dimensionierung eines Variablen-Feldes können Sie sich einen bestimmten Speicherbereich reservieren und ansprechen. Ein Variablen-Feld kann aber nur sehr begrenzt für einfachste Strukturen eingesetzt werden. Denken Sie nur an die Problematik der unterschiedlichen Typen der Datenfelder. Die Hilfe naht, wie könnte es auch anders sein, wieder in Form von Library-Routinen. Gleich drei verschiedene Libraries stellen Routinen zur Speicherbelegung zur Verfügung. Daran sehen Sie bereits, wie wichtig die Speicherreservierung für die Handhabung der Libraries ist. Zuerst wenden wir uns dem Manager der Software, der Exec-Library zu. Mit der folgenden Routine wird ein Speicherbereich in der benötigten Größe reserviert:

Format: `memoryBlock=AllocMem(byteSize,requirements)`

Die *requirements* finden Sie wieder im Anhang. Wenn Sie keine Verwendung mehr für den Speicherbereich haben, sollten Sie ihn unbedingt wieder freigeben. Das ist deshalb so wichtig, da der Amiga nicht merkt, wer ihm da ein Stück vom Speicher abgezwackt hat. Ein nicht freigegebener Speicherbereich bleibt verloren. Nur ein Neustart des Computers bringt ihn wieder zurück.

Format: `FreeMem(memoryBlock,byteSize)`

Wenn Sie mehrere Speicherblöcke benötigen, müssen Sie natürlich alle einzeln freigeben. Bei der zweiten Library mit Belegungs-Routinen, der Intuition, ist die Freigabe der reservierten Speicher einfacher. In einem Gedächtnis-Schlüssel (RememberKey) werden alle reservierten Speicherbereiche notiert:

Format: `memoryBlock=AllocRemember(RememberKey,Size,Flags)`

Die Freigabe eines oder mehrerer Speicherbereiche erfolgt durch nachstehende Routine:

Format: `FreeRemember(RememberKey,ReallyForget)`

Das folgende Programm gibt ein einfaches Gadget auf den Bildschirm aus. Für die Gadget-Struktur wird ein Speicherbereich benötigt, in den dann die einzelnen Datenfelder geschrieben werden. Die Reservierung und die Freigabe des Speichers erfolgt mit

den beiden zuletzt genannten Routinen der Intuition. Das Gadget selbst hat in dem Programmbeispiel keine besondere Funktion. Wenn Sie mit der Maus das Gadget anklicken, so wird das Programm beendet.

```
REM Aus  Pfad: Darstellung/6System/Aus
'P6-1 **Beispiel für AllocRemember und FreeRemember**
DEFINT a-z
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION AddGadget&() LIBRARY
DECLARE FUNCTION RemoveGadget&() LIBRARY
LIBRARY ":bue/intuition.library"
w=WINDOW(2)/3+3:h=WINDOW(3)/2-11:a=-1
art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
mem&=AllocRemember&(rk&,5Ø,art&)
IF mem&=Ø THEN PRINT "Speicher zu klein":BEEP:GOTO ende
LOCATE 12,29:PRINT "?"
Gadget
gad&= AddGadget&(WINDOW(7),mem&,-1)
CALL OnGadget&(mem&,WINDOW(7),Ø)
WHILE a
  IF PEEKW(mem&+12)<127 THEN a=Ø
WEND
CALL OffGadget&(mem&,WINDOW(7),Ø)
egad&= RemoveGadget&(WINDOW(7),mem&)
ende:
IF rk& THEN CALL FreeRemember&(rk&,-1)
LIBRARY CLOSE:SYSTEM

SUB Gadget STATIC
SHARED mem&,w%,h%
  POKEL mem&,Ø
  POKEW mem&+4,w%  :POKEW mem&+6,h% :POKEW mem&+8,4Ø
  POKEW mem&+1Ø,2Ø :POKEW mem&+12,129
  POKEW mem&+14,1  :POKEW mem&+16,1
END SUB
```

Nachdem *AllocRemember* eine Funktion ist, wird sie zu Programmbeginn als solche festgelegt. Da sämtliche Routinen des Programms aus der Intuition-Library stammen, lag es natürlich nahe, auf die Speicherreservierung dieser Library zurückzugreifen, auch wenn nur ein einziger Speicherbereich benötigt wird. Für die *Flags* erhielt die Va-

riable *art&* MEMF__PUBLIC (1) + MEMF__CHIP (2) + MEMF__CLEAR (65536) zugewiesen. Nach der Reservierung wird mit IF mem&=0 geprüft, ob die Routine ausgeführt werden konnte. Wenn nicht, wird das Programm beendet. Wurde das Gadget selektiert, wird der Speicherbereich wieder freigegeben und das Programm beendet. Sie haben sicherlich bereits entdeckt, daß mit *IF rk&* vorher geprüft wurde, ob die Variable logisch wahr, also von Null verschieden ist. Eine solche Prüfung ist notwendig und immer empfehlenswert, da bei einem Fehler zum Label »ende« verzweigt wird. Würde ein nicht reservierter Speicher freigegeben, so wäre der Absturz des Rechners nicht zu verhindern.

6.6 Abfragen des IDCMP

Viele Leckerbissen der Grafik, wie HAM oder Super-Bitmaps, können wir nur programmieren, indem wir eigene Screens und Windows programmieren. Wie Sie später sehen werden, können diese nicht mit den normalen Befehlen des Basic erstellt werden. Sie unterliegen damit nur begrenzt dem Zugriff des Basic-Interpreters. Das bedeutet, daß dadurch auch keine Mitteilungen an das Programm gegeben werden können. Ein solches Programm könnte also nur stur, ohne irgendeine Eingriffsmöglichkeit, abgespult werden. Warum ist das so?

Einfach ausgedrückt liegt es daran, daß für den Amiga das Computer-System keine Einheit darstellt. Es besteht aus einer Reihe von Geräten, wie Tastatur, Bildschirm, Diskettenlaufwerk, Drucker etc. Für die Verknüpfung zwischen der Software und der Hardware sorgen die Devices. Eines der Devices ist das Input Device. Es regelt alle Ein- und Ausgabeaktivitäten des Amiga. Immer wenn Sie die Tastatur bedienen, die Maus bewegen oder einen Mausknopf drücken, erkennt das das Input Device und bildet ein Eingabeereignis in Form einer System-Message. Diese Ereignisse können über Message-Ports empfangen werden.

Das Einrichten und Abfragen über einen Message-Port nimmt uns normalerweise der Basic-Interpreter ab. Mit kräftiger Unterstützung der Intuition können aber auch wir an diese Nachrichten gelangen. Die Intuition stellt die Exec-Messages in einem »Intuition Direct Communications Message-Port« (IDCMP) zur Verfügung. Beim Anlegen eines neuen Fensters (wird später erklärt) geben wir an, welche Art von Nachrichten wir empfangen wollen. Bei aktiviertem Window finden wir dann an der Speicherstelle 86 der Window-Struktur einen Zeiger auf den Anwender-Message-Port. Aus diesem Port können wir nun die Nachrichten, die wir benötigen, auslesen. Dieses Auslesen des *port* geht mit einer Library-Routine der exec.library recht einfach vonstatten:

Format: `message=GetMsg(port)`

Liegt keine Nachricht vor, so erhält *message* den Wert Null. Ansonsten zeigt *message* auf eine Struktur (Anhang), aus der die einzelnen Nachrichten ausgelesen werden können. Das Wichtigste jedoch ist, daß jede empfangene Nachricht unbedingt zurückgegeben werden muß, da sonst der Message-Port mit der letzten Nachricht blockiert bleibt. Für die Rückgabe der Nachricht rufen Sie folgende Routine auf:

Format: `ReplyMsg(port)`

Das alles hört sich sicherlich sehr kompliziert an. Daß sich der Programmieraufwand in Grenzen hält, sehen Sie an dem folgenden Beispiel. Es zeigt die Abfrage der Tastatur an Hand eines kleinen Textprogrammes. Beim Ausprobieren werden Sie feststellen, daß kein Cursor zu sehen ist. Trotzdem ist sogar eine Fehlerkorrektur möglich. Am Ende einer Zeile wird sogar nach einem (normallangen) Wort auf die nächste Zeile geschaltet.

```
REM IOtext  Pfad: Darstellung/6System/IOtext
'P6-2 **Beispiel für GetMsg und ReplyMsg**
DEFINT a-z
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION OpenWindow&() LIBRARY
DECLARE FUNCTION GetMsg&() LIBRARY
LIBRARY ":bue/exec.library"
LIBRARY ":bue/intuition.library"
LIBRARY ":bue/graphics.library"
Fehler=Ø :CALL initMemory :IF Fehler THEN ende
CALL WindowStruktur :win&=OpenWindow&(mem&)
IF win&=Ø THEN Fehler=-1:GOTO ende
rp&=PEEKL(win&+5Ø):CALL SetAPen&(rp&,1)
a=-1 :x=2:y=2Ø
WHILE a
  text-="" :me&=PEEKL(win&+86)
  holen: mess&=GetMsg&(me&):IF mess&=Ø THEN holen
  Cla&=PEEKL(mess&+2Ø)     'Class of IntuitionMessage
  cod=PEEKW(mess&+24)      'Code of IntuitionMessage
  CALL ReplyMsg&(mess&)
  IF Cla&=2Ø97152& THEN
    IF cod = 8 AND x>9 THEN x=x-8
    IF cod = 13 THEN x=2:y=y+8:GOTO Sprung
```

```
    IF cod<32 THEN Sprung
    IF cod>127 AND cod<161 THEN Sprung
    text-=CHR-(cod)
    CALL Move&(rp&,x,y):CALL text&(rp&,SADD(text-),1)
    IF x>52Ø AND cod=32 OR x>52Ø AND cod=45 THEN x=-6:y=y+8
    x=x+8:IF x>6ØØ THEN x=2:y=y+8
    Sprung: IF y>1ØØ THEN a=Ø
  END IF
WEND
ende:
IF win& THEN CALL CloseWindow&(win&)
IF rk& THEN CALL FreeRemember&(rk&,-1)
IF Fehler THEN BEEP:PRINT "Fehler bei den Libraries"
LIBRARY CLOSE:END

SUB initMemory STATIC
  SHARED rk&,mem&,Fehler%
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  mem&=AllocRemember&(rk&,56,art&)
  IF mem&=Ø THEN Fehler%=-1
END SUB

SUB WindowStruktur STATIC
SHARED mem&
  scr&=PEEKL(WINDOW(7)+46)
  tit-="RETURN mehrmals druecken beendet das Programm!"+CHR-(Ø)
  POKEW mem&+4,64Ø:POKEW mem&+6,2ØØ
  POKE mem&+8,3    :POKE mem&+9,1
  POKEL mem&+1Ø,2Ø97152&  'IDCMP-Flag VANILLAKEY(2Ø97152)
  POKEL mem&+14,4Ø96      'Fenster-Flag AKTIVATE(4Ø96)
  POKEL mem&+26,SADD(tit-)
  POKEL mem&+3Ø,scr&:POKEW mem&+38,1ØØ :POKEW mem&+4Ø,1Ø
  POKEW mem&+42,638 :POKEW mem&+44,25Ø :POKEW mem&+46,1
END SUB
```

Werfen wir nun, wie gewohnt, einen Blick auf das Programm. Zu den Funktions-Deklarationen gehört natürlich auch *GetMsg*. Die exec.library benötigen wir zum Auslesen des Message-Ports, die intuition.library für das Fenster und die graphics.library für die Text-Routinen. Das Unterprogramm *initMemory* reserviert Speicherplatz für die Window-Struktur, wie im letzten Kapitel besprochen. Das zweite Unterprogramm *WindowStruktur* versorgt die NewWindow-Struktur (siehe Anhang) mit den gewünschten Parametern. Dabei sind für dieses Programm die IDCMP-Flags von Interesse. Es wird

nur das Flag für VANILLAKEY gesetzt. Damit werden die Tastaturbewegungen als ASCII-Codes an den Message-Port gesendet.

Nach dem Öffnen des Fensters mit *OpenWindow* holen wir uns die Adresse des Rast-Ports aus der Window-Struktur. In der WHILE/WEND-Schleife wird es interessant. Hier läuft das Programm ab. Der *port* wird aus der Window-Struktur geholt und in der Variablen *me&* gespeichert. Bei »holen« wird mit *GetMsg* der Message-Port abgefragt. Ist das Ergebnis <>0, so liegt eine Nachricht an. In *Cla&* wird die Class und in *cod%* der Code der Intuition-Message gemerkt. Nun wird mit *ReplyMsg* die Message zurückgegeben. Der folgende IF-Block wird vom Programm durchlaufen, wenn eine Meldung vom Typ VANILLAKEY (2097152) vorliegt. In dem IF-Block werden nach den empfangenen ASCII-Codes in *cod%* die Zeichen gesetzt, der Cursor positioniert, auf eine neue Zeile gesetzt und bei Überschreitung eines bestimmten Y-Wertes der Schleifenzähler unwahr. Mit den üblichen Aufräumungsarbeiten wird das Programm beendet.

Mit der gleichen Methode können Sie auch die Mausbewegungen abfragen oder die Betätigung einer Maustaste registrieren. Sie sehen, die Abfrage des IDCMP ist nur etwas ungewohnt, aber sicherlich nicht besonders kompliziert.

6.7 Die Intuition, Mittler zwischen Mensch und Maschine

Die Intuition ist für den Basic-Programmierer die wichtigste Library der System-Software. Ohne sie kämen wir nicht einmal dazu, das Basic des Amiga aufzurufen. Schließlich müssen wir vorher von der Workbench aus starten. Und der Workbench-Screen wird bereits von der Intuition verwaltet. Alle Grafik-Ausgabeelemente, mit denen wir es zu tun haben, werden von der Intuition verwaltet. Sie sorgt dafür, daß wir uns nicht mit den Routinen der unteren Software-Ebenen und der Hardware herumzuschlagen brauchen.

Die wichtigste Aufgabe der Intuition ist die Verwaltung der Screens und der Windows. Dazu gehören auch die Routinen für die zugehörigen Kommunikations-Elemente Requester, Gadgets, Menüs, der Mauszeiger und noch einige andere mehr. Es ist daher nicht verwunderlich, daß sich auch das Amiga-Basic fleißig dieser Routinen bedient.

6.7.1 Custom-Screens

Wie man einen neuen Bildschirm unter Basic erstellt, wissen Sie inzwischen längst. Ich erwähnte bereits, daß es jedoch einige Anwendungsfälle gibt, und das sind die interessantesten, bei denen wir mit einem Basic-Screen nicht auskommen. Sie stehen dann vor dem Problem, selbst einen Intuition-Screen einzurichten. Keine Sorge, das hört sich dramatischer an, als es ist. Die Intuition selbst stellt uns die Routinen zur Verfügung,

die die Arbeit relativ einfach machen. Um einen eigenen Screen anzulegen, gehen Sie am besten in der folgenden Reihenfolge vor:

- Reservieren eines Speicherbereiches für die NewScreen-Struktur.
- Eintragen der gewünschten Parameter für die NewScreen-Struktur in den reservierten Speicherbereich.
- *OpenScreen* aufrufen, die Funktion gibt einen Zeiger auf die Struktur des neuen Screens zurück.
- Der reservierte Speicherbereich wird nicht mehr benötigt und kann an das System zurückgegeben werden.

Die Routine *OpenScreen* legt aus den Parametern der NewScreen-Struktur einen neuen Bildschirm an. Neben den vorgegebenen Werten des Anwenders werden eine Reihe weiterer Parameter gesetzt und alle Ausgabeelemente der unteren Software-Ebene erstellt. Wir kommen auf dieses Thema bei den View-Modi noch einmal zu sprechen. In dem Augenblick, ab dem der neue Screen existiert, wird die Hilfs-Struktur *NewScreen* nicht mehr benötigt.

Die NewScreen-Struktur finden Sie wieder im Anhang. Außerdem werden in diesem Buch etliche Programme mit Custom-Screens ausgerüstet, so daß Sie genügend Beispiele zur Verfügung haben. In einen auf diese Art erstellten Anwender-Bildschirm können Sie mit den Betriebssystem-Routinen zeichnen oder auch Texte ausgeben.

6.7.2 Custom-Windows

Eine Kommunikation mit dem Anwender ist jedoch mit einem neuen Screen alleine noch nicht möglich. Dazu müssen Sie zusätzlich ein neues Fenster mit den Intuition-Routinen öffnen. Ein solches neues Fenster können Sie auch ohne einen neuen Screen anlegen. Sie setzen dazu für den Typ das Bit für WBENCHSCREEN. Die NewWindow-Struktur und die komplette Window-Struktur mit allen Flags finden Sie im Anhang. Die Reihenfolge der einzelnen Arbeitsgänge, die notwendig sind, um ein neues Fenster anzulegen, sind ähnlich wie bei einem neuen Screen:

- Einen Speicherbereich für die NewWindow-Struktur reservieren.
- Die gewünschten Parameter für die NewWindow-Struktur in den reservierten Speicherbereich poken.
- Die Routine *OpenWindow* aufrufen. Mit diesem Aufruf legt die Intuition die wesentlich umfangreichere Window-Struktur an. Den Zeiger auf das neue Fenster liefert Ihnen die Funktion.
- Der reservierte Speicherbereich wird nicht mehr benötigt und sollte freigegeben werden.

Damit haben Sie das Wichtigste zum Anlegen neuer Anwender-Screens und Windows erfahren. Die Einzelheiten können Sie aus den einzelnen Programmen und den Strukturen im Anhang entnehmen.

6.8 Die einfachen Grafik-Routinen

In diesem Kapitel finden Sie eine Zusammenfassung der einfachen Grafik-Routinen. In einigen werden Sie die gleichen Resultate wiedererkennen, die Ihnen ein normaler Basic-Befehl auch gebracht hätte. Das ist nicht verwunderlich. Wie bereits kurz erwähnt, werden diese Routinen auch vom Amiga-Basic selbst aufgerufen. Dabei sind diese Library-Routinen für Ihre Programme nicht überflüssig. Zum ersten ergibt sich oft ein Geschwindigkeitsvorteil, wenn Sie die Maschinensprache-Routinen direkt anspringen. Bei einem normalen Basic-Befehl muß der Basic-Interpreter immer erst in einer Liste nachsehen, bevor er die Routinen aufrufen kann. Zum zweiten können Sie mit diesen Routinen direkt in den Screen (bzw. in die Bitmap) schreiben. Was das bedeutet und welche Vorteile Sie bei Ihren Programmen dadurch erlangen, werden Sie im Verlauf des Buches reichlich erfahren.

Die folgenden Routinen sind nicht im Detail erklärt. Sie finden daher alle im Buch verwendeten Library-Routinen im Anhang wieder. Dort ist alles, was zum Einsatz der entsprechenden Routine notwendig ist, in Kurzform zusammengefaßt. Die wichtigsten Gesichtspunkte, besonders die Typen der einzelnen Variablen, können Sie jedoch den jeweiligen Programmbeispielen entnehmen. Denken Sie daran, daß vorher die *graphics.library* geöffnet sein muß. Funktionen deklarieren Sie vorher mit DECLARE FUNCTION ... LIBRARY. Andernfalls erhalten Sie die Fehlermeldung »overflow« des Basic-Interpreters. Viele Beispiele sind Einfachstprogramme, die Ihnen nur die Funktion und den Aufruf der Routinen demonstrieren. Am Ende des Kapitels finden Sie jedoch die meisten Routinen in größerem Zusammenhang zu einem flotten Zeichenprogramm zusammengefaßt. Zur Ausführung der Grafik-Routinen der unteren Ebene benötigen Sie sehr oft die Adresse der RastPort-Struktur. Diese liefert Ihnen die Funktion WINDOW(8).

6.8.1 Die Zeichen-Routinen

Die erste Funktion setzt einen Bildpunkt bei der Koordinate *x, y*.

Format: `Ergebnis=WritePixel(rastPort,x,y)`

Werden die Grenzen des RastPorts überschritten, so liefert Ergebnis den Wert –1, sonst 0. Das Programmbeispiel setzt 61 Linien, bestehend aus Reihen von Bildpunkten bis zum Fensterrand. Beim Überschreiten der Fenstergrenzen wird das Ergebnis –1 und die innere Schleife verlassen.

```
REM Punkt  Pfad: Darstellung/6System/Punkt
'P6-3
DECLARE FUNCTION WritePixel&() LIBRARY
LIBRARY ":bue/graphics.library"
DEFINT a-z:FOR x1=Ø TO 6Ø:x=Ø:y=Ø:erg=Ø
WHILE erg=Ø:erg=WritePixel&(WINDOW(8),x,y)
x=x+x1:y=y+1:WEND:NEXT:LIBRARY CLOSE :END
```

Die Farbkennung des Pixels bei der Koordinate (x,y) liefert die zweite Routine als Ergebnis.

Format: `Ergebnis=ReadPixel(rastPort,x,y)`

Das folgende Kurzprogramm durchsucht Punkt für Punkt einen rechteckigen Bildschirmbereich (*x*y*). Jeder gefundene Grafikpunkt, der logisch wahr ist (also nicht in der Hintergrundfarbe Null gesetzt ist), wird mit *WritePixel* an drei versetzte Positionen gezeichnet. Dadurch entstehen drei Kopien des mit Zufallslinien gezeichneteten Bild-Originals.

```
REM Kopie  Pfad: Darstellung/6System/Kopie
'P6-4
DECLARE FUNCTION WritePixel&() LIBRARY
DECLARE FUNCTION ReadPixel&() LIBRARY
LIBRARY ":bue/graphics.library":DEFINT a-z:w=WINDOW(2)
h=WINDOW(3):x=w/6:y=h/2: FOR i = Ø TO 99 :fa=RND*3
x1=RND*x:y1=RND*y :x2=RND*x:y2=RND*y
LINE (x1,y1)-(x2,y2),fa:NEXT:w1=w/4:w2=w1*3
FOR y3=Ø TO y:FOR x3= Ø TO x
  fa=ReadPixel&(WINDOW(8),x3,y3):COLOR fa
  FOR d=w1 TO w2 STEP w1 :IF fa THEN
  erg=WritePixel&(WINDOW(8),x3+d,y3)
  END IF:NEXT d
NEXT x3,y3 :COLOR 1:LIBRARY CLOSE :END
```

Die nächste Routine setzt den grafischen Cursor auf jede gewünschte Bildschirmposition.

Format: `Move(rastPort,x,y)`

Das Beispielprogramm setzt auf die Grafikzeilen 8 bis 80 das Schriftzeichen "_". Probieren Sie ruhig einmal aus, ob Sie das mit den Befehlen des Standard-Basic auch schaffen.

```
REM Stufe  Pfad: Darstellung/6System/Stufe
'P6-5
LIBRARY ":bue/graphics.library"
FOR y% = 8 TO 8Ø: x%=x%+8
CALL Move&(WINDOW(8),x%,y%)
PRINT "_":NEXT
LIBRARY CLOSE :END
```

Zum Zeichnen einer Linie dient die folgende Routine. Vorher muß mit *Move* der grafische Cursor auf die Start-Koordinate der Linie gebracht werden.

Format: `Draw(rastPort,x,y)`

Das Beispiel zeigt eine hübsche grafische Spielerei mit den Routinen *Move* und *Draw*.

```
REM Spielerei  Pfad: Darstellung/6System/Spielerei
'P6-6
LIBRARY":bue/graphics.library":DEFINT a-z
w=WINDOW(2)/2:h=WINDOW(3)/2:FOR x=Ø TO WINDOW(2) STEP 8
COLOR fa:fa=fa+1:IF fa>3 THEN fa=1
FOR y=WINDOW(3) TO Ø STEP -8
CALL Move&(WINDOW(8),w,h):CALL Draw&(WINDOW(8),x,y)
NEXT y,x:LIBRARY CLOSE:COLOR 1:END
```

Auch das Zeichnen von Kreisen oder Ellipsen ist recht einfach mit den Routinen der Libraries zu bewerkstelligen.

Format: `DrawEllipse(rastPort,x,y,horizRadius,vertRadius)`

Wenn Sie einen der ersten Amiga 1000 besitzen, so befindet sich diese Routine nicht in der *graphics.library*. Sie sollten sich daher die neueste Version von einem Bekannten besorgen.

Soll ein Kreis gezeichnet werden, so werden für *horizRadius* und *vertRadius* die gleichen Werte als Radius eingesetzt. Das gilt aber nur für einen Screen mit niedriger Auflösung. Bei einem hochauflösenden Bildschirm kann als Näherungswert (wie das folgende Beispiel zeigt) der Y-Radius als halber horizontaler Radius eingesetzt werden.

```
REM Ellipsen und Kreise  Pfad: Darstellung/6System/Ellips
'P6-7
LIBRARY ":bue/graphics.library"
DEFINT a-z:w=WINDOW(2):h=WINDOW(3)
FOR i = Ø TO 99 :fa=RND*3:COLOR fa
x1=RND*w/3:y1=RND*(h-3Ø):xr=RND*w/6:yr=RND*h/6
CALL DrawEllipse&(WINDOW(8),x1,y1,xr,yr):NEXT
FOR i = Ø TO 99 :fa=RND*3:COLOR fa
x1=(RND*w/3)+(w/2):y1=RND*(h-3Ø):xr=RND*w/6:yr=xr/2
CALL DrawEllipse&(WINDOW(8),x1,y1,xr,yr):NEXT
COLOR 1:LIBRARY CLOSE :END
```

Die folgende Routine zählt zu den Besten aus der *graphics.library.* Mit ihr können Sie ein beliebiges Polygon mit einer wahnsinnigen Geschwindigkeit zeichnen. Unter Polygon ist bei dieser Routine ein beliebiges Vieleck mit sich kreuzenden Linien, also eine echte Liniengrafik, zu verstehen. *PolyDraw* ist dank seiner Geschwindigkeit so vielseitig einsetzbar, daß Ihnen wahrscheinlich erst nach und nach weitere Anwendungen einfallen werden.

Format: `PolyDraw(rastPort,count,array)`

Als *array* wird der Zeiger auf ein Datenfeld übergeben, das die einzelnen Koordinaten der Polygons (x und y) enthält. Der Zähler *count* enthält die Anzahl der Eckpunkte. Das nachstehende Programm stellt meine Behauptungen unter Beweis. Dank der bereits mehrfach erwähnten Geschwindigkeit zeigt es eine animierte Liniengrafik.

```
REM Kran  Pfad: Darstellung/6System/Kran
'P6-8
DEFINT a-z :DIM d(1Ø3),e(1Ø3),we(255)
FOR i=Ø TO 255:we(i)=RND*127:NEXT:WAVE Ø,we
h=PEEKW(PEEKL(WINDOW(7)+46)+14)
Richtung=1:Start=5:Ende=h-99
LIBRARY ":bue/graphics.library"
SCREEN 2,32Ø,h,1,1
WINDOW 2,"Taste=Aktion Return=ende",,,2
FOR i = Ø TO 1Ø3:READ d(i):e(i)=d(i):NEXT
```

```
CALL Move&(WINDOW(8),d(0),d(1))
CALL PolyDraw&(WINDOW(8),10,VARPTR(d(0)))
Ablauf:
IF Richtung = 1 THEN Richtung=-1 ELSE Richtung=1
SWAP Start,Ende
FOR h = Start TO Ende STEP Richtung
  FOR i= 21 TO 103 STEP 2:e(i)=d(i)+h:NEXT
  LINE (220,45)-(280,WINDOW(3)),0,bf
  SOUND 20,2.1,155,0
  CALL Move&(WINDOW(8),250,20)
  CALL PolyDraw&(WINDOW(8),42,VARPTR(e(20)))
NEXT
taste:ta-=INKEY-:IF ta-="" THEN taste
IF ta-<>CHR-(13) THEN Ablauf
SCREEN CLOSE 2:WINDOW CLOSE 2:LIBRARY CLOSE
ERASE d,e,we:END
DATA 75,140,10,140,10,180,100,180,100,140,70,170
DATA 50,160,250,5,260,10,100,140
DATA 250,40,280,60,280,80,280,85,220,85,220,80
DATA 230,80,230,76,228,76,228,67,235,62,265,62
DATA 272,67,228,67,230,69,230,72,245,72,245,69,255,69
DATA 255,72,270,72,270,69,230,69,270,69,272,67
DATA 272,76,270,76,270,80,260,80,260,76,240,76,240,80
DATA 230,80,230,76,270,76,270,80,280,80,220,80
DATA 220,60,280,60,220,60,250,40
```

Die Koordinaten des Polygons werden aus den Datas in das Feld *d()* eingelesen. Die ersten 10 Koordinatenpaare zeichnen einen einfachen Kran. Im zweiten Teil, ab dem Label »Ablauf«, findet die Animation statt. Der PKW als Kranlast wird ab der Feldvariablen *d(20)* gezeichnet. Damit die Auf- und Abwärtsbewegung reibungslos vonstatten geht, wird zu den Y-Koordinaten die Höhe *h* addiert und in dem Feld *e()* zwischengespeichert. Obwohl diese Berechnung einige Zeit dauert, wird in der Hauptschleife die Bewegung fast flimmerfrei gezeichnet. Bevor die Liniengrafik in einer neuen Position gezeichnet wird, löscht eine einfache LINE-Anweisung die bestehende Grafik. Bei jedem neuen Durchlauf werden die entsprechenden Variablen *Start* und *Ende* getauscht und der STEP-Zähler positiv oder negativ. Damit kann die gleiche Schleife für den entgegengesetzten Bewegungsablauf eingesetzt werden.

6.8.2 Rund um die Farben

6.8.2.1 Wie die Zeichenfarbe gesetzt wird

Wie Sie bereits im ersten Abschnitt, im Kapitel über die Farben, gelesen haben, stellt der Amiga drei verschiedene Zeichenstifte zur Verfügung. Die Bezeichnung der entsprechenden Library-Routinen finden Sie in der folgenden Zusammenstellung:

- A-Pen für Zeichnungen und Textausgabe.
- B-Pen für Texthintergrund.
- O-Pen für den Rand von Flächen.

Die ersten beiden Stifte werden mit den folgenden Routinen aufgerufen:

Format: `SetAPen(rastPort,pen)`
Format: `SetBPen(rastPort,pen)`

In dem folgenden Beispiel wird die Anwendung der beiden Routinen demonstriert:

```
REM AundB  Pfad: Darstellung/6System/AundB
'P6-9
LIBRARY ":bue/graphics.library":DEFINT a-z:b=2
rp&=WINDOW(8):w=WINDOW(2)/3:h=WINDOW(3)/2
start: a=-1:tt-="ergrundfarbe: "
WHILE a
  fa=RND*2+1:CALL Move&(rp&,w,h)
  IF b=2 THEN
    CALL SetAPen&(rp&,fa):t-="Vord"+tt-+CHR-(48+fa)
  ELSE
    CALL SetBPen&(rp&,fa):t-="Hint"+tt-+CHR-(48+fa)
  END IF
  CALL Text&(rp&,SADD(t-),19)
  ta-=INKEY-:IF ta-<>"" THEN a=Ø:b=b-1
WEND
IF b THEN start
CALL SetAPen&(rp&,1):CALL SetBPen&(rp&,Ø)
LIBRARY CLOSE :END
```

Für den dritten Stift, den *O-Pen*, gibt es (noch) keine Library-Routine. Durch eine Änderung der RastPort-Struktur können Sie sich aber behelfen. Dazu müssen die Datenfelder 27 für die Randfarbe und 32 für die Flags (AREAOUTLINE = 8) abgeändert werden. Sie dürfen nur nicht vergessen, den ursprünglichen Zustand anschließend

wiederherzustellen, da Sie sonst die komplette RastPort-Struktur durcheinanderbringen. Das Unterprogramm aus dem folgenden Programmbeispiel setzt den *O-Pen* und löscht ihn auch wieder. Sie können es unverändert für Ihre eigenen Programme übernehmen.

```
REM moderne Kunst  Pfad: Darstellung/6System/modern
'P6-1Ø
DEFINT a-z :h=PEEKW(PEEKL(WINDOW(7)+46)+14)
SCREEN 1,32Ø,h,3,1:WINDOW 2,,,Ø,1
x=WINDOW(2)-2:y=WINDOW(3)-2:a=-1
WHILE a
  fa=RND*5+2: CALL SetOPen(fa)
  FOR eck=Ø TO 15: AREA (RND*x,RND*y):NEXT
  CLS:AREAFILL:CALL SetOPen(Ø)
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
WINDOW CLOSE 2:SCREEN CLOSE 1:END
SUB SetOPen(OFa%) STATIC
rp&=WINDOW(8)
IF PEEKW(rp&+32) AND 8 THEN
  POKEW rp&+32,PEEKW(rp&+32)AND NOT 8:POKE rp&+27,255
ELSE
  POKE rp&+27,OFa%:POKEW rp&+32,PEEKW(rp&+32) OR 8
END IF
END SUB
```

Das Programm zeichnet ein Vieleck mit 16 Ecken. Die Koordinaten der Eckpunkte werden durch Zufallszahlen festgelegt. Die Randfarbe ändert sich bei jedem Bild. Gefüllt wird es zur Demonstration des *O-Pens* immer mit der Farbe Weiß. Der erste Aufruf des Unterprogrammes mit dem Parameter *fa* setzt die Randfarbe *fa*. Der zweite Aufruf, der nur den Pseudowert Null mitbekommt, stellt den normalen Zustand wieder her.

Fertigen Sie bitte keine Hardcopies von den einzelnen Grafiken. Sie könnten sonst glatt als moderner Künstler entdeckt werden. Neben den Zeichenstiften, die die Farben zum Zeichnen, für den Text oder den Rand einer Fäche festlegen, können Sie auch noch bestimmen, wie die einzelnen Punkte auf den Bildschirm gebracht werden. Es stehen Ihnen 4 Modi zur Verfügung, die Sie auch noch miteinander kombinieren können:

- JAM1 (Wert 0) ersetzt einen Pixel durch die Farbe des APens.
- JAM2 (Wert 1) für Linien- oder Füllmuster von PATTERN.
- COMPLEMENT (Wert 2) stellt einen Bildpunkt revers dar.
- INVERSIVID (Wert 4) für inverse Textdarstellung.

In dem folgenden Programm wird der Modus COMPLEMENT eingesetzt. Durch diesen Modus ist eine Bildverschiebung möglich, die den Bildhintergrund nicht zerstört.

```
REM Schiebung  Pfad: Darstellung/6System/Schiebung
'P6-11
LIBRARY ":bue/graphics.library":DEFINT a-z:DIM t&(2)
a=-1: rp&=WINDOW(8):w=WINDOW(2):h=WINDOW(3):x=w/3:y=h/2
tt-="Diesen Text können Sie mit ":t&(Ø)=SADD(tt-)
tt-="den Cursor-Tasten bewegen. ":t&(1)=SADD(tt-)
tt-="RETURN beendet das Programm":t&(2)=SADD(tt-)
FOR i=Ø TO 1Ø:LINE (w*RND,h*RND)-(w*RND,h*RND),3*RND,bf:NEXT
CALL SetDrMd&(rp&,2)
WHILE a
  FOR i=Ø TO 2: CALL Move&(rp&,x,y+(i*8))
    CALL Text&(rp&,t&(i),27):NEXT
  xa=x:ya=y:wert=Ø
  taste:ta-=INKEY-:IF ta-="" THEN taste
  IF ta-=CHR-(13) THEN a=Ø
  IF ta->CHR-(27) AND ta-<CHR-(32) THEN wert=ASC(ta-)-27
    IF wert=4 THEN x=x-2 :IF x<Ø THEN x=Ø
    IF wert=3 THEN x=x+2 :IF x>w-256 THEN x=w-256
    IF wert=2 THEN y=y+1 :IF y>h-18 THEN y= h-18
    IF wert=1 THEN y=y-1 :IF y<8 THEN y=8
  FOR i=Ø TO 2: CALL Move&(rp&,xa,ya+(i*8))
    CALL Text&(rp&,t&(i),27):NEXT
WEND
CALL SetDrMd&(rp&,1):LIBRARY CLOSE :ERASE t&:END
```

Als Hintergrund werden zuerst einige Rechtecke gezeichnet. Anschließend wird der Zeichenmodus 2 aufgerufen. Das Programm läuft in der WHILE/WEND-Schleife ab. In ihr werden zuerst die drei Textzeilen ausgegeben. Nun wartet das Programm auf die Betätigung der Cursor-Tasten. Je nach Taste werden die Werte für x oder y erhöht oder verringert. Gleichzeitig werden die neuen Werte auf eine Überschreitung der Fenstergrenzen überprüft und gegebenenfalls wird die Änderung des Wertes rückgängig gemacht. Daraufhin wird der Text zum zweiten Mal an der gleichen Position ausgegeben. Durch den Modus COMPLEMENT verschwindet der Text und kann beim nächsten Schleifendurchlauf auf die neue Position gesetzt werden.

6.8.2.2 Die Farbgebung

Nachdem Sie nun die Library-Routinen für die Festlegung der Zeichen- und Randfarbe und den Modus kennen, fehlen nur noch die Routinen, die sich mit der Farbgebung direkt befassen. Auch hier finden Sie neue Wege, die die Programmierung erheblich erleichtern und beschleunigen.

Wenn Sie mit den Befehlen des Standard-Basic die Zusammensetzung einer Farbe wissen wollen, so müssen Sie sich auf's Raten beschränken. Damit ist es jetzt vorbei. Zwei Möglichkeiten stehen Ihnen offen, die Farbwerte zu ermitteln. Die Farben sind in Wortlänge in der ColorMap-Struktur (siehe Anhang) abgelegt. Die vier Nibbles (Halbbytes) des Wortes enthalten die Farbbestandteile wie folgt:

```
ØRGB
```

Sie brauchen also nur die Farbe aus der Struktur lesen und die einzelnen Farbbestandteile zu isolieren. Den Weg über die einzelnen Strukturen können Sie jedoch mit einer Library-Routine etwas abkürzen:

Format: `Farbwort=GetRGB4(colormap,entry)`

Zum Setzen einer Farbe kennen Sie die PALETTE-Anweisung des Standard-Basic. Diese Anweisung ist jedoch nicht besonders anwenderfreundlich. Oder wissen Sie auswendig, welche Werte für die einzelnen Farbbestandteile eingegeben werden müssen? Die folgende Routine enthebt Sie dieses Problems:

Format: `SetRGB4(viewPort,index,r,g,b)`

In einem Farb-Wort sind für jeden Farbanteil die Werte von 0–15 möglich. Diese Kombination aus 16*16*16 Farbbestandteilen ergibt ja bekanntermaßen die 4096 verschiedenen Farben des Amiga. Genauso gehen Sie bei der Routine *SetRGB4* vor. Für jeden der einzelnen Farbbestandteile *r*, *g* und *b* werden Werte zwischen 0 und 15 eingegeben. Einfacher gehts wirklich nicht.

Die beiden Routinen *SetRGB4* und *GetRGB4* sind in dem folgenden Programm eingesetzt. Es zeigt die Farbbestandteile der gewählten Farbe in den Werten von 0 bis 15 an. Unter Eingabe dieser Werte kann auch die aktuelle Farbe geändert werden. Nach jeder Änderung eines Farbbestandteiles wird die aktuelle Farbmischung angezeigt. Mit einem Druck auf eine beliebige Taste wird das Programm beendet.

```
REM Bunt  Pfad: Darstellung/6System/Bunt
'P6-12 **Beispiel für GetRGB4 und SetRGB4**
DEFINT a-z
INPUT "Bitte Farbtiefe: ",t:t=ABS(t):IF t>5 THEN END
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):sw=32Ø:tiefe=t
DECLARE FUNCTION ViewPortAddress&() LIBRARY
DECLARE FUNCTION GetRGB4&() LIBRARY
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"
SCREEN 1,sw,sh/2,tiefe,1
tit-="F1=Farbe holen   F2=Farbe aendern"
WINDOW 2,tit-,,Ø,1 :rp&=WINDOW(8)
w=WINDOW(2):h=WINDOW(3):fa=1:max=2^tiefe-1
vp&=ViewPortAddress&(WINDOW(7))
farben fa:zeigen fa
taste:ta-=INKEY-:IF ta-="" THEN taste
IF ta-=CHR-(129) THEN Farbeholen
IF ta-=CHR-(13Ø) THEN Farbesetzen
WINDOW CLOSE 2:SCREEN CLOSE 1:LIBRARY CLOSE: END

Farbeholen:
LOCATE 1,1:INPUT "Bitte Farbnummer ",fa
fa=ABS(fa):IF fa>max THEN BEEP: GOTO Farbeholen
farben fa:zeigen fa
GOTO taste

Farbesetzen:
t-(Ø)="Bitte Rotanteil:   ":t-(1)="Bitte Gruenanteil: "
t-(2)="Bitte Blauanteil:  ":c(1)=g:c(2)=b
FOR i=Ø TO 2
  LOCATE 1,1:PRINT t-(i);:INPUT c(i)
  c(i)=ABS(c(i)):IF c(i)>15 THEN BEEP:GOTO Farbesetzen
  CALL SetRGB4&(vp&,fa,c(Ø),c(1),c(2))
  LOCATE 1,1:PRINT SPACE-(25):zeigen fa
NEXT
GOTO taste

SUB farben (fa%) STATIC
  SHARED rp&,w%,h%,max%
  CALL SetRast&(rp&,fa%) :w%=w%-1:h%=h%-1
  FOR i% = Ø TO max% STEP 2
    br%=INT((w%)/(max%+1)):COLOR i%
```

```
    AREA (br%*i%,56):AREA (br%*i%,h%)
    AREA ((i%+2)*br%,56):AREAFILL
    COLOR i%+1
    AREA (i%*br%,h%):AREA ((i%+2)*br%,56)
    AREA ((i%+2)*br%,h%):AREAFILL
    NEXT :COLOR 1
  LOCATE 3,1:PRINT "Aktuelle Farbe: ";:PRINT USING"##";fa%
  LOCATE 4,1:PRINT STRING-(18,"=")
END SUB

SUB zeigen (fa%) STATIC
  SHARED vp&,r%,g%,b%
  cm&=PEEKL(vp&+4):F%=GetRGB4&(cm&,fa%)
  r%=INT(F%/2^8):Rest%=F% MOD 2^8
  LOCATE 5,1:PRINT "Rotanteil:      ";:PRINT USING"##";r%
  g%=INT(Rest%/2^4)
  LOCATE 6,1:PRINT "Gruenanteil:    ";:PRINT USING"##";g%
  b%=Rest% MOD 2^4
  LOCATE 7,1:PRINT "Blauanteil:     ";:PRINT USING"##";b%
END SUB
```

Gleich zu Beginn kann der Anwender eine Farbtiefe von 1 bis 5 eingeben. Entsprechend dieser Tiefe wird nach den Öffnungs-Routinen für die Library ein Bildschirm mit niedriger Auflösung geöffnet. Nachdem auch das Fenster eingerichtet ist, wird mit *ViewPortAddress* die ViewPort-Adresse an die Variable *vp&* übergeben. Anschließend wird das erste Unterprogramm »farben« aufgerufen. In ihm wird mit der Grafik-Routine *SetRast* der Bildschirm mit einer wählbaren Farbe eingefärbt.

Format: `SetRast(rastPort,color)`

Darauf zeichnet das Programm mit AREA-Anweisungen Dreiecke in sämtlichen zur Verfügung stehenden Farben.

Das zweite Unterprogramm »zeigen« holt sich zuerst aus der ViewPort-Struktur die Adresse der ColorMap (*cm&*). Nun kann mit *GetRGB4* das gewünschte Farbwort (*F%*) geholt werden. Aus diesem Farbwort werden die einzelnen Farbbestandteile *r%, g%* und *b%* isoliert und auf dem Bildschirm ausgegeben.

Kehren wir nun wieder zum Hauptprogramm zurück. Beim Label »taste« wartet das Programm auf eine Tastatureingabe des Anwenders. Bei einem Druck auf die Taste F1 verzweigt das Programm zu »Farbeholen«, bei F2 springt es zu »Farbesetzen« und eine andere Taste beendet das Programm. Bei der Sprungmarke »Farbeholen« wird die

gewünschte Farbe eingegeben und mit der eingegebenen Farbnummer die beiden Sub-Routinen aufgerufen. Bei »Farbesetzen« werden nacheinander die drei Farbwerte eingegeben und sofort mit der Routine *SetRGB4* in die Farbtabelle des Amiga geschrieben.

Das kleine Programm hat Ihnen sicherlich gezeigt, wie einfach die beiden Farb-Routinen einzusetzen sind. Das einfache Setzen von einzelnen Farben ist ja ganz nett. Doch was machen Sie, wenn Sie zum Beispiel alle 32 Farben ändern wollen? Wer kennt nicht die ellenlangen PALETTE-Anweisungen am Anfang eines Basic-Programmes. Auch *SetRGB4* würde in einem solchen Fall einen erheblichen Aufwand bedeuten. Mit einer einzigen Grafik-Routine und 32 DATAs können wir effektvoll das Problem vom Tisch fegen.

Format: `LoadRGB4(viewPort,colormap,count)`

Bei *colormap* wird ein Zeiger auf ein Datenfeld, bestehend aus kurzen Ganzzahlen (Worten), übergeben. Der *count* erhält die Anzahl der Farben ohne den Wert Null. Das einfache und blitzschnelle Laden einer neuen Farbtafel ist nur ein Gesichtspunkt, der für die Routine spricht. Durch die erreichte Geschwindigkeit beim Setzen einer neuen Farbtafel sind auch schnelle Farb-Animationen möglich. Das Programm »Rolltreppe« gibt Ihnen eine kleine Kostprobe dieser Möglichkeit.

```
REM Rolltreppe  Pfad: Darstellung/6System/Rolltr
'P6-13  **Beispiel für LoadRGB4 und Farb-Animation**
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):sw=64Ø:tiefe=3
DECLARE FUNCTION ViewPortAddress&() LIBRARY
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"
SCREEN 1,sw,sh,tiefe,2
WINDOW 2,,,Ø,1 :w=WINDOW(2):h=WINDOW(3)
vp&=ViewPortAddress&(WINDOW(7))
FOR i=Ø TO 7:READ f(i):NEXT
CALL LoadRGB4&(vp&,VARPTR(f(Ø)),8)
fa=2:x=w/2-5Ø:d=1ØØ:a=-1
FOR y = Ø TO h
  LINE (x,y)-(x+d,y),fa
  x=x-1:d=d+2:fa=fa+1:IF fa>7 THEN fa=2
NEXT
```

```
WHILE a
  f(8)=f(2)
  FOR i = 2 TO 7:f(i)=f(i+1):NEXT
  CALL LoadRGB4&(vp&,VARPTR(f(Ø)),8)
  taste:ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
WINDOW CLOSE 2:SCREEN CLOSE 1:LIBRARY CLOSE: END

DATA &hØØ9Ø,&hØfff,&hØ444,&hØ555
DATA &hØ666,&hØ777,&hØ888,&hØ999
```

Nach den Öffnungs-Routinen wird mit *ViewPortAddress* die für die Routine *LoadRGB4* benötigte ViewPort-Adresse geholt. Die einzelnen Farb-Worte werden in das Variablen-Feld *f()* eingelesen. Damit kann nun die Routine *LoadRGB4* aufgerufen werden. In der folgenden FOR/NEXT-Schleife wird eine Treppe aus den Farben 2–7 gezeichnet. Aus den Datenwerten können Sie ersehen, daß es sich dabei um verschiedene Grautöne handelt.

In der WHILE/WEND-Schleife läuft nun die Farb-Animation ab. Dazu wird zuerst die Farbe 2 in die Feld-Variable *F(8)* übertragen. In der anschließenden Schleife werden die einzelnen Farbwerte um eine Position nach unten geschoben. Mit den neuen Werten wird mit Hilfe von *LoadRGB4* die neue Farbtafel gesetzt. Dieser Vorgang wiederholt sich bei jedem Schleifendurchlauf. Die Farben 2 bis 7 rotieren also. Durch diese Rotation der Farben entsteht die Bewegung der Treppe.

Sicher fallen Ihnen auf Anhieb eine Menge weiterer Möglichkeiten ein, um durch die Farb-Rotation aus toten Grafiken lebendige Bilder zu zaubern.

6.8.3 Gefüllte Flächen

Bisher haben Sie erst eine Routine zum Füllen von Flächen kennengelernt. Mit *SetRast* haben wir den kompletten Bildschirm mit einer Farbe ausgemalt. Genauso einfach zeichnen wir ein gefülltes Rechteck:

Format: `RectFill(rastPort,xl,yl,xu,yu)`

Probieren wir die Routine gleich in einem kleinen Beispiel aus. In dem Programm »Quadrat« werden in einem gleichmäßigen Ablauf 177 gefüllte Quadrate optisch recht eindrucksvoll gezeichnet.

```
REM Quadrat  Pfad: Darstellung/6System/Quadrat
'P6-14 **Beispiel für RectFill**
DEFINT a-z:
LIBRARY ":bue/graphics.library"
scr&=PEEKL(WINDOW(7)+46):rp&=scr&+84
FOR i= Ø TO 176
  y1=99-i/2 :y2=1Ø9+i/2
  CALL RectFill&(rp&,31Ø-i,y1,33Ø+i,y2)
NEXT
LIBRARY CLOSE :END
```

Jetzt werden wir ein Problem beim Schopfe packen, das Ihnen wahrscheinlich schon etliches Kopfzerbrechen bereitet hat. Ich spreche vom Füllen einer beliebig gezeichneten Fläche in jeder gewünschten Farbe. Wenn Sie im Programmablauf eine Figur zeichnen, können Sie mit dem Parameter »Rand« der PAINT-Anweisung eine andere Farbe zum Füllen verwenden. In dem Augenblick, wo der Anwender selbst ins Geschehen eingreifen kann, zum Beispiel bei einem Zeichenprogramm, ist es jedoch mit der Herrlichkeit vorbei. Oder wissen Sie in einem Zeichenprogramm, nach etlichen Zwischenschritten, in welcher Farbe genau ein Polygon etc. gezeichnet wurde? Bei sich überschneidenden Linien von unterschiedlichen Farben ist dann sowieso nichts mehr zu retten. Das ganze Bild läuft Ihnen mit der gewählten Farbe voll. Das wird wohl auch der Hauptgrund sein, warum die meisten Zeichenprogramme eine UNDO-Funktion anbieten. Dabei wird der aktuelle Zustand zwischengespeichert, damit man bei dem aufgeführten Dilemma das alte Bild wieder hervorzaubern kann. Das kostet natürlich viel Speicherplatz, der für die normalen Zeichenmöglichkeiten verlorengeht.

Bei der Lösung des Problems müssen wir in zwei Schritten vorgehen. Wir sind einmal unlogisch und sehen uns den zweiten Schritt zuerst an. Die Grafik-Routine *Flood* füllt eine beliebige Fläche in der eingestellten Farbe:

Format: `Flood(rp,mode,x,y)`

Ausschlaggebend für den Füllvorgang ist *mode.* Wird dafür der Wert 0 eingesetzt, so gilt der Outline-Modus. Beginnend bei den Koordinaten *x* und *y* sucht das System in allen Richtungen nach einem Grafikpunkt, der der Randfarbe entspricht. Beim Color-Modus mit dem Wert 1 für *mode* wird dagegen das Pixel bei der Position *x* und *y* zur Farbgebung verwendet. Alle Pixel mit der gleichen Farbe werden auf die neue Farbe eingefärbt. Damit ist bei einer geschlossenen Fläche ein Auslaufen des Bildes ausgeschlossen!

Das wäre natürlich prima, wenn das so einfach wäre. Versuchen Sie bitte nicht, die Routine ohne weitere Vorbereitung einzusetzen. Das wäre nämlich das beste Mittel, um

Ihren Amiga abstürzen zu lassen. Die Routine *Flood* erwartet einen eigenen Speicherbereich, von dem aus die Farbe in das aktuelle Bild (sprich RastPort) kopiert wird. Dieser Speicherbereich nennt sich TmpRas. Die Größe hängt von der zu füllenden Fläche ab. Im Zweifelsfall reserviert man die komplette Bildschirmgröße. *TmpRas* ist unabhängig von der gewählten Farbtiefe. Es muß daher nur eine Ebene reserviert werden. Alles was zur Initialisierung von *TmpRas* gehört, habe ich in einer kleinen Subroutine zusammengefaßt. Sie können sie daher in ähnlicher Form in Ihren eigenen Programmen einsetzen. Welche Parameter die Routine erwartet, entnehmen Sie bitte den Zeilen unter »Vorbereitungen«.

```
Vorbereitungen:
DECLARE FUNCTION AllocRaster&() LIBRARY
DECLARE FUNCTION AllocRemember&() LIBRARY
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"
art&=3+(2^16):rek&=0:rk&=VARPTR(rek&)
'b=Screen-Breite
'h=Screen-Höhe
'tiefe=Farbtiefe
scr&=PEEKL(WINDOW(7)+46) 'Screen-Adresse
rp&=scr&+84              'RastPort-Adresse
mb&=PEEKL(rp&+4)         'Bitmap-Adresse

InitTemp:
bmBytPerRow=b/8 'Bytes per Grafikzeile
volum&=bmBytPerRow*h
  bp&=AllocRaster&(b,h)
  IF bp&=0 THEN Fehl=-1:RETURN
  CALL BltClear&(bp&,volum&,0)
'Fuer Struktur TmpRas
mb&=AllocRemember&(rk&,20,art&)
POKEL mb&,bp&
POKEL mb&+4,volum&
CALL InitTmpRas&(mb&,bp&,volum&)
POKEL rp&+12,mb&
RETURN
```

Zu Beginn von »InitTemp« berechnen wir die benötigten Bytes und speichern sie in der Variablen *volum&*. Mit der Grafik-Routine *AllocRaster* reservieren wir für eine Bitplane den Speicher. Dann reservieren wir mit *AllocRemember* einen Speicherbereich für die TmpRas-Struktur und versorgen sie mit den benötigten Daten. Anschließend können wir *TmpRas* initialisieren:

Format: `InitTmpRas(tmpras,buffer,size)`

Zum Schluß poken wir noch die Adresse der TmpRas-Struktur in die Speicherstelle 12 der RastPort-Struktur. Uff, das wäre geschafft. Das Beste ist, wir probieren das neue Wissen gleich an einem hübschen Spiel aus.

```
REM MusicMind  Pfad: Darstellung/6System/MusicMind
'P6-15
'**Beispiel für InitTmpRas und Flood**
CLEAR
IF FRE(-1)<12ØØØØ& THEN PRINT "Speicher zu klein":END
DEFINT a-z:DIM f(15),fr!(11),xp(11),yp(11),ton(1Ø)
DECLARE FUNCTION AllocRaster&() LIBRARY
DECLARE FUNCTION AllocRemember&() LIBRARY
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"
art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)

tiefe=4 :b=64Ø:h=256: SCREEN 1,b,h,tiefe,2
WINDOW 2,,,Ø,1:scr&=PEEKL(WINDOW(7)+46)
rp&=scr&+84 :mb&=PEEKL(rp&+4):vp&=scr&+44
FOR i=Ø TO 15:READ f(i):NEXT       'Farben
CALL LoadRGB4&(vp&,VARPTR(f(Ø)),16)
GOSUB InitTemp :IF Fehl THEN ende
FOR i=Ø TO 11:READ fr!(i),xp(i),yp(i):NEXT  'Frequenz,Position

start:
IF punkte>tarek THEN tarek=punkte
LOCATE 1,1:PRINT "Tagesrekord: "tarek
LOCATE 3.1:PRINT "Punkte:      "punkte: punkte = Ø
eing: LOCATE 5,1:PRINT  "Schwierigkeit(1-5): ";
t-=INPUT-(1):IF t-<CHR-(49) OR t->CHR-(53) THEN eing
sw=VAL(t-) :LOCATE 5,14:PRINT sw "   "
da!=(1Ø-sw)*.75:max=sw+5
FOR x=17Ø TO 62Ø STEP 15Ø
  CALL SetAPen&(rp&,1)
  CALL RectFill&(rp&,x,Ø,x+1Ø,253)
NEXT x
FOR y=Ø TO 248 STEP 62
  CALL SetAPen&(rp&,2)
  CALL RectFill&(rp&,181,y,619,y+5)
NEXT y
```

```
FOR p=Ø TO 11
  CALL SetAPen&(rp&,p+4)
  CALL Flood&(rp&,1,xp(p),yp(p))
NEXT

ablauf:
FOR i=Ø TO max
  FOR j= Ø TO i'anz
    RANDOMIZE TIMER:ton(j)=INT(RND*12)
    CALL SetAPen&(rp&,3)
    CALL Flood&(rp&,1,xp(ton(j)),yp(ton(j)))
    SOUND fr!(ton(j)),da!,15Ø,1
    CALL SetAPen&(rp&,ton(j)+4)
    CALL Flood&(rp&,1,xp(ton(j)),yp(ton(j)))
  NEXT j
  LOCATE 7,1:PRINT SPACE-(8):n=Ø
  FOR j= Ø TO i
    muss=MOUSE(Ø):WHILE MOUSE(Ø)<=Ø :WEND
    muss=MOUSE(Ø):posx=MOUSE(3):posy=MOUSE(4)
    IF posx < (xp(ton(j))-7Ø) OR posx > (xp(ton(j))+7Ø) THEN j=i
    IF posy < (yp(ton(j))-3Ø) OR posy > (yp(ton(j))+3Ø) THEN j=i
    n=n+1
  NEXT j
  LOCATE 7,1
  IF j=n THEN
    PRINT "OK     ":punkte=punkte+i*sw
  ELSE
    PRINT "falsch" :punkte=punkte-sw
  END IF
  LOCATE 3,14:PRINT punkte
NEXT
LOCATE 1Ø,1:PRINT "Neustart? j/n";
t-=INPUT-(1)
IF t-="j" THEN LOCATE 1Ø,1:PRINT SPACE-(14):GOTO start

ende:
IF bp& THEN CALL FreeRaster&(bp&,b,h)
IF rk& THEN CALL FreeRemember&(rk&,-1)
WINDOW CLOSE 2:SCREEN CLOSE 1
IF Fehl THEN :BEEP:BEEP:PRINT "Speicher zu klein!"
ERASE f,fr!,xp,yp,ton
LIBRARY CLOSE :END
```

```
InitTemp:
bmBytPerRow=b/8 'Bytes per Grafikzeile
volum&=bmBytPerRow*h
  bp&=AllocRaster&(b,h)
  IF bp&=Ø THEN Fehl=-1:RETURN
  CALL BltClear&(bp&,volum&,Ø)
'Fuer Struktur TmpRas
mb&=AllocRemember&(rk&,2Ø,art&)
POKEL mb&,bp&
POKEL mb&+4,volum&
CALL InitTmpRas&(mb&,bp&,volum&)
POKEL rp&+12,mb&
RETURN

Farbwerte:
DATA &HØddd,&HØØØØ,&hØ999,&hØfØ3
DATA &HØff2,&hØfcØ,&hØfaØ
DATA &HØE9Ø,&hØc87,&HØa87
DATA &hØØfØ,&hØØcØ,&hØØ9Ø
DATA &hØØØf,&hØØØc,&hØØØa
FrequenzenPositionen:
DATA 164.81,245,32,185.3,395,32,196,545,32
DATA 22Ø,245,94,246.94,395,94,261.63,545,94
DATA 293.66,245,156,329.63,395,156,37Ø.7,545,156
DATA 392,245,218,44Ø,395,218,493.Ø8,545,218
```

Die Öffnungsroutinen haben wir bereits besprochen. Anschließend lesen wir mit *LoadRGB4* eine neue Farbtafel ein, initialisieren den *TmpRas* und lesen weitere Daten ein. Bei der Sprungmarke »start« beginnen die Vorbereitungen. Nach der Ausgabe der Spielpunkte gibt der Anwender die gewünschte Schwierigkeitsstufe ein.

In den drei folgenden FOR/NEXT-Schleifen wird ein Liniengerüst (zur Demonstration von *Flood* mit verschiedenen Farben) gezeichnet und mit *Flood* im Modus 1 gefüllt. Jedes Feld erhält dabei eine andere Farbe (und läuft doch nicht aus).

Bei »ablauf« geht es erst richtig los. Die Variable *max* erhält die Anzahl der Durchgänge, die von der eingestellten Schwierigkeit abhängen. Das einzelne Spiel innerhalb eines Durchganges hängt von dem Schleifenzähler *i* der äußeren Schleife ab. Das bedeutet, im ersten Durchgang muß nur ein Ton mit Farbfeld erkannt werden. Mit jedem Durchgang erhöht sich die Melodie um einen Ton. Der Durchgang wird also von Spiel zu Spiel schwieriger. Die erste innere FOR/Next-Schleife zwingt mit RANDOMIZE TIMER den Zufallsgenerator zu einem neuen Anfangswert. Dann kann der Ton aus

12 möglichen Tönen mit RND ermittelt werden. Die diesem Ton zugehörenden X- und Y-Koordinaten und der Frequenzwert (sie wurden aus den DATAs eingelesen) werden in den folgenden Anweisungen benötigt.

Zuerst wird mit *SetAPen* die aktuelle Farbe auf rot eingestellt. Mit *Flood* wird das zugehörige Feld neu gefüllt. Dabei erklingt der Zufalls-Ton. Die Zeichenfarbe wird wieder zurückgeändert und das Farbfeld in der richtigen Farbe eingefärbt. Kurz gesagt, es erklingt ein Ton und das zugehörige Feld blitzt kurz rot auf.

In der zweiten inneren FOR/NEXT-Schleife muß der Anwender die Tonfolge nachvollziehen. Die gespeicherten Werte in *xp()* und *yp()* überwachen, ob das richtige Feld angeklickt wurde. Beim ersten Fehler wird die Schleife verlassen. Stimmt die Anzahl der Schleifendurchläufe nicht mit den Soll-Durchläufen überein, wird ein Fehler ausgegeben. Der Rest bedarf keiner besonderen Erklärung. Mit dem kleinen Spiel werden Sie sicherlich viel Freude haben.

6.9 Ein ungewöhnliches Zeichenprogramm

Wie zu Beginn des Kapitels versprochen, folgt nun ein Programm, das die meisten der besprochenen Library-Routinen in einem größeren Zusammenhang zeigt. Am besten starten Sie gleich das Programm. Es meldet sich mit einem gespiegelten Titelbild. Ein Druck auf die linke Maustaste zaubert ein leeres Fenster herbei. Ein weiterer Druck, dieses Mal auf die Menü-Taste, zeigt, daß zwei Menüs zur Auswahl stehen. Das Projekt-Menü bietet nur zwei Optionen an. Mit *ENDE* beenden Sie das Programm und mit *Neubeginn* löschen Sie das Fenster und starten von vorne.

Interessanter ist da schon das Werkzeuge-Menü. Haben Sie schon eine Funktion ausgewählt? Dann haben Sie sicher verblüfft festgestellt, daß aus dem Mauspfeil plötzlich ein Fadenkreuz geworden ist. Exakt in der Mitte befindet sich der heiße Punkt. Sie werden feststellen, daß Sie damit sehr genau zeichnen können. Nun aber zu den einzelnen Optionen im Überblick.

- **Farben:** Es öffnet sich ein Fenster, in dem Sie eine von 32 Farben selektieren können. Die aktuelle Farbe wird durch ein Rechteck umrandet.
- **Füllen:** Jede umrandete Fläche, egal aus wieviel Farben die Umrandung besteht, wird in der gewählten Farbe gefüllt.
- **Freihand:** Solange die linke Maustaste gedrückt ist, werden Punkte gesetzt.
- **Linie:** Die Linie beginnt an der Position, an der die Maustaste gedrückt wurde, und endet an der Position, an der diese losgelassen wird. Währenddessen ist die Linie frei über dem Hintergrund verschiebbar.

- **Rechteck:** Es wird ein Rechteck gezeichnet, ansonsten Funktion wie Linie.
- **Kreis:** Der Mausklick markiert den Mittelpunkt. Ein Ziehen der Maus in eine beliebige Richtung vergrößert oder verkleinert den Kreis, bis die Taste losgelassen wird.
- **Ellipse:** Auch hier markiert der Mausklick den Mittelpunkt. Die beiden Ellipsen-Radien werden exakt nach den Maus-Koordinaten, bezogen auf den Mittelpunkt, gezeichnet. Bis die Taste losgelassen wird, können auch hier sämtliche Variationen von Ellipsen über dem Hintergrund verschoben werden.
- **Polygon:** Jeder Mausklick setzt einen Eckpunkt des Polygons und verbindet ihn mit dem vorhergegangenen Eckpunkt mit einer Linie. Die letzte Linie, die das Polygon schließt, wird mit
- **PolyEnde** aufgerufen.
- **Radieren:** Wie Rechteck. Das Rechteck wird jedoch in der Hintergrundfarbe gefüllt und radiert damit innerhalb der Rechteck-Umrandung alles aus.

Damit ist bereits das Repertoire an Optionen erschöpft. Sie können das Programm aber, wenn Sie Lust dazu haben, noch um einige Funktionen erweitern.

Die Überschrift zu diesem Kapitel lautet *Ein ungewöhnliches Zeichenprogramm.* Was ist denn so ungewöhnlich daran? Nun, zum einen arbeitet das Programm für ein Basic-Programm sehr schnell (außer der Option *Freihand*). Außerdem ist ein Auslaufen des Bildes nahezu ausgeschlossen. Und last not least, sämtliche Zeichen-Routinen, Farbgebungen, der Zeichen-Modus und die Konstruktion des Mauszeigers werden durch Routinen der Libraries ausgeführt.

6.9.1 Library-PAINT

```
REM LibraryPAINT.V1.Ø  Pfad: Darstellung/6System/LibsPAINT
'P6-16
CLEAR
IF FRE(-1)<17ØØØØ& THEN PRINT "Speicher zu klein!":END
DEFINT a-z:DIM f(31),sd(72)
DECLARE FUNCTION TextLength&() LIBRARY
DECLARE FUNCTION WritePixel&() LIBRARY
DECLARE FUNCTION AllocRaster&() LIBRARY
DECLARE FUNCTION AllocRemember&() LIBRARY
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"

art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
m1&=AllocRemember&(rk&,1ØØ,art&):tiefe=5 :b=32Ø
h=PEEKW(PEEKL(WINDOW(7)+46)+14): SCREEN 1,b,h,tiefe,1
```

```
WINDOW 2,,,16,1
scr&=PEEKL(WINDOW(7)+46):rp&=scr&+84 :vp&=scr&+44
win&=WINDOW(7)
farben: FOR i=Ø TO 31:READ f(i):NEXT
CALL LoadRGB4&(vp&,VARPTR(f(Ø)),32)
MausZeiger:
FOR i=Ø TO 71 STEP 2
  READ sd:POKEW m1&+i,sd AND 65535&
NEXT i
GOSUB InitTemp :IF fehl THEN ende

MENU 1,Ø,1,"P R O J E K T  " :MENU 1,1,1,"Neubeginn      "
MENU 1,2,1,"E N D E        " :MENU 2,Ø,1,"WERKZEUGE      "
MENU 2,1,1,"  Farben       " :MENU 2,2,1,"  Fuellen      "
MENU 2,3,1,"  Freihand     " :MENU 2,4,1,"  Linie        "
MENU 2,5,1,"  Rechteck     " :MENU 2,6,1,"  Kreis        "
MENU 2,7,1,"  Ellipse      " :MENU 2,8,1,"  Polygon      "
MENU 2,9,1,"  PolyEnde     " :MENU 2,1Ø,1,"  radieren     "
ON MENU GOSUB menuekontr:MENU ON
ON MOUSE GOSUB mauskontr:MOUSE ON

start:
MENU OFF:MOUSE OFF:CALL ClearPointer&(win&)
Titel
MENU ON:MOUSE ON
farbe=1:fehl=Ø:merkx=-1:zeig=-1:clzeig=Ø
a=-1:bb=Ø
WHILE a
  IF fehl THEN a=Ø:bb=Ø
WEND
IF bb THEN start

ende:
IF bp& THEN CALL FreeRaster&(bp&,b,h)
IF rk& THEN CALL FreeRemember&(rk&,-1)
WINDOW CLOSE 2:SCREEN CLOSE 1
IF fehl THEN BEEP:BEEP:PRINT "Speicher zu klein!"
ERASE f:MENU RESET :CALL ClearPointer&(win&)
LIBRARY CLOSE :END
menuekontr:
IF MENU(Ø)=1 THEN
  ON MENU(1) GOTO neu,aus
END IF
```

```
altnum=num :num=MENU(1):GOSUB PointerPruefen
IF num=1 THEN fen
IF num=9 THEN PolyEnd
MENU 2,num,2:MENU 2,altnum,1
RETURN

PointerPruefen:
IF num>1 THEN
  IF zeig=-1 THEN CALL
SetPointer&(win&,m1&,15,15,-11,-9):zeig=Ø:clzeig=-1
ELSE
  IF clzeig THEN CALL ClearPointer&(win&):zeig=-1:clzeig=Ø
END IF
RETURN

mauskontr:
Position
x1=x2:y1=y2
ON num GOTO
blind,fuell,Hand,Linien,Block,Kreis,Ellipse,Polygon,blind,radier

blind:RETURN

neu:a=Ø:bb=-1:CLS:RETURN
fen:
WINDOW 3,"Auswahl durch Mausklick",(Ø,1Ø)-(252,35),Ø,1
FOR i=Ø TO 31: CALL SetAPen&(rp&,i)
CALL RectFill&(rp&,8*i+2,25,8*(i+1)+2,4Ø) :NEXT
Rechteck 8*farbe,24,8*(farbe+1)+2,41
muss=MOUSE(Ø)
waehlen: IF MOUSE(Ø)=Ø THEN waehlen
x=MOUSE(3):WINDOW CLOSE 3
farbe =INT(x/8):IF farbe>31 THEN farbe=31
CALL SetAPen&(rp&,farbe)
warte:IF MOUSE(Ø)<Ø THEN warte
num=altnum :GOTO PointerPruefen

markieren:
CALL SetDrMd&(rp&,3)
WHILE MOUSE(Ø)<>Ø
  Position
  IF bl THEN
    Rechteck x1,y1,x2,y2 :Rechteck x1,y1,x2,y2
```

```
  ELSE
    Linie x1,y1,x2,y2 :Linie x1,y1,x2,y2
  END IF'
WEND
CALL SetDrMd&(rp&,1)
RETURN

Hand:
WHILE MOUSE(Ø)<>Ø
  Position :erg%=WritePixel&(rp&,x2,y2)
WEND
IF erg% THEN fehl=-1
RETURN

Linien:
GOSUB markieren :Linie x1,y1,x2,y2
RETURN

Block:
bl=-1:GOSUB markieren:bl=Ø:Rechteck x1,y1,x2,y2
RETURN

Kreis:
CALL SetDrMd&(rp&,3)
WHILE MOUSE(Ø)<>Ø
  Position
  r=ABS(SQR((x2-x1)^2+(y2-y1)^2))
  CALL DrawEllipse&(rp&,x1,y1,r,r)
  CALL DrawEllipse&(rp&,x1,y1,r,r)
WEND
CALL SetDrMd&(rp&,1)
CALL DrawEllipse&(rp&,x1,y1,r,r)
RETURN

Ellipse:
CALL SetDrMd&(rp&,3)
WHILE MOUSE(Ø)<>Ø
  Position
  xr=ABS(x2-x1):yr=ABS(y2-y1)
  CALL DrawEllipse&(rp&,x1,y1,xr,yr)
  CALL DrawEllipse&(rp&,x1,y1,xr,yr)
WEND
CALL SetDrMd&(rp&,1)
```

```
CALL DrawEllipse&(rp&,x1,y1,xr,yr)
RETURN

Polygon:
IF merkx<Ø THEN merkx=x1:merky=y1:xanf=x1:yanf=y1
Position
  Position:CALL Move&(rp&,merkx,merky)
  CALL Draw&(rp&,x2,y2):merkx=x2:merky=y2
RETURN

PolyEnd:
  CALL Move&(rp&,merkx,merky):CALL Draw&(rp&,xanf,yanf)
  merkx=-1:num=altnum
RETURN

fuell:
CALL Flood&(rp&,1,x1,y1)
RETURN

radier:
bl=-1:GOSUB markieren:bl=Ø: CALL SetAPen&(rp&,Ø)
IF x1>x2 THEN SWAP x1,x2
IF y1>y2 THEN SWAP y1,y2
CALL RectFill&(rp&,x1,y1,x2,y2):CALL SetAPen&(rp&,farbe)
RETURN
aus:a=Ø:RETURN

InitTemp:
bmBytPerRow=b/8 'Bytes per Grafikzeile
volum&=bmBytPerRow*h
  bp&=AllocRaster&(b,h)
  IF bp&=Ø THEN fehl=-1:RETURN
  CALL BltClear&(bp&,volum&,Ø)
mb&=AllocRemember&(rk&,2Ø,art&)  'Fuer Struktur TmpRas
POKEL mb&,bp&
POKEL mb&+4,volum&
CALL InitTmpRas&(mb&,bp&,volum&)
POKEL rp&+12,mb&
RETURN

SUB Titel STATIC
SHARED b,h
```

```
text1-="L I B R A R Y   P A I N T":tt1&=SADD(text1-)
text2-=" by Horst-Rainer Henning ":tt2&=SADD(text2-)
SCREEN 2,b,h,1,1: WINDOW 3,,,Ø,2
scr2&=PEEKL(WINDOW(7)+46)
rp2&=scr2&+84 :vp2&=scr2&+44
CALL SetRGB4&(vp2&,Ø,15,7,4)
tl=TextLength&(rp2&,tt1&,25)
txp=5Ø:typ1=1ØØ:typ2=11Ø:sy1=9Ø:sy2=115:i=sy2+5Ø
CALL Move&(rp2&,txp,typ1):CALL Text&(rp2&,tt1&,25)
CALL Move&(rp2&,txp,typ2):CALL Text&(rp2&,tt2&,25)
spiegelnX:
FOR z=sy1 TO sy2
  i=i-1
  CALL ClipBlit&(rp2&,txp,z,rp2&,txp,i,tl,1,192)
NEXT
muss=MOUSE(Ø):WHILE MOUSE(Ø)=Ø:WEND
WINDOW CLOSE 3:SCREEN CLOSE 2
END SUB

SUB Position STATIC
  SHARED x2,y2
  muss=MOUSE(Ø):x2=MOUSE(1):y2=MOUSE(2)
END SUB

SUB Rechteck(x1%,y1%,x2%,y2%) STATIC
  SHARED rp&
  CALL Move&(rp&,x1%,y1%)
  CALL Draw&(rp&,x2%,y1%)
  CALL Draw&(rp&,x2%,y2%)
  CALL Draw&(rp&,x1%,y2%)
  CALL Draw&(rp&,x1%,y1%)
END SUB

SUB Linie(x1%,y1%,x2%,y2%) STATIC
  SHARED rp&
  CALL Move&(rp&,x1%,y1%)
  CALL Draw&(rp&,x2%,y2%)
END SUB

Farbwerte:
DATA &hØfff,&hØbbb,&hØ888,&hØØØØ,&hØfØf,&hØffØ
DATA &hØdd5,&hØfbØ,&hØf9Ø,&hØe9Ø,&hØc9Ø,&hØ99Ø
DATA &hØ9ØØ,&hØaØØ,&hØcØØ,&hØdØØ,&hØeØØ,&hØfØØ,&hØf89
```

```
DATA &h0090,&h00a0,&h00c0,&h00d0,&h00e0,&h00f0
DATA &h0009,&h000a,&h000c,&h000d,&h000e,&h000f,&h030f

zeiger:
DATA &h0,&h0,&H440,&HC60,&H440,&HC60,&H440,&HC60
DATA &h0440,&h0c60,&H440,&HFC7E,&HFC7E,&HFC7E
DATA &h0,&h0,&H100,&H0,&H0,&H0,&HFC7E,&HFC7E
DATA &h0440,&hfc7e,&H440,&HC60,&H440,&HC60
DATA &h0440,&h0c60,&H440,&HC60,&H0,&H0,&H0,&H0
```

6.9.2 Beschreibung

Nach den üblichen Zeilen zum Öffnen der Libraries wird ein Speicherbereich für die neuen Daten des Mauszeigers reserviert (*m1&*). Ein Screen mit Window werden geöffnet und die Adreßzeiger der benötigten Strukturen ermittelt. Anschließend werden die Daten der Farbtafel eingelesen und mit *LoadRGB4* gesetzt. In den reservierten Speicherbereich *m1&* werden die Sprite-Daten des Mauszeigers gepoket. Es schließt sich der Sprung zu »InitTemp« an, den wir bereits besprochen haben.

Den Menü-Anweisungen folgen die Aktivierungen der Unterbrechungsfähigkeit des Menüs und der Maus. Beim Label »start« wird das Unterprogramm »Titel« aufgerufen. Neben der reinen Textausgabe mit *Move* und *Text* erfolgt dort eine Spiegelung. Diese werden wir in einem der nächsten Kapitel besprechen. Das Programm wartet nun in der WHILE/WEND-Schleife, bis ein Fehler auftritt oder ein Unterbrechungsereignis (Maus, Menü) eintritt.

Kommen wir nun zu den einzelnen Subroutinen. Die Sprungmarke »menuekontr« wird bei jeder Betätigung der Menü-Taste (rechte Maustaste) angesprungen. Hier befindet sich quasi die Schaltzentrale für die einzelnen Menü-Optionen. Wurde das Menü 0 ausgewählt, so wird sofort zu den Subroutinen »neu« und »aus« verzweigt. Die alte Menü-Nummer wird in der Variablen *altnum* gemerkt und die neue Nummer an *num* übergeben. Die beiden Nummern für die Farbwahl und das Ende des Polygons werden direkt abgearbeitet. Die letzte Zeile setzt mit *num* den neuen und entfernt mit *altnum* den alten Menü-Haken. Die Nummern 2 bis 8 werden also nur registriert, aber nicht bearbeitet. Bei »PointerPruefen« wird bei jeder Menü-Nummer größer 1 der grafische Mauszeiger mit *SetPointer* konstruiert. Bei der Nummer 1 (Farbwahl) ruft die Routine *ClearPointer* den Standard-Mauspfeil zurück. Die beiden Zeiger *zeig* und *clzeig* sorgen dafür, daß diese Arbeit unterbleibt, wenn schon die richtigen Zeiger ausgegeben sind.

Beim Label »mauskontr«, zu dem bei jeder Betätigung der linken Maus-Taste verzweigt wird, werden nun die einzelnen Zeichen-Jobs zugewiesen. Zuerst wird mit dem Unterprogramm »Position« die aktuelle Position des Maus-Zeigers geholt. Die beiden Varia-

blen *x2* und *y2* gelten für die anderen Subroutinen als End-Koordinaten und müssen daher in die Start-Koordinaten *x1* und *y1* umgewandelt werden. Nun kann anhand der Menü-Nummern zum Abarbeiten der einzelnen Zeichen-Routinen verzweigt werden.

Für die Auswahl der Farben ist die Subroutine »fen« zuständig. Zuerst wird ein schmales Fenster geöffnet. In der FOR/NEXT- Schleife zeichnet *RectFill* 32 Rechtecke in den zur Verfügung stehenden Farben. Die aktuelle *farbe* wird im Unterprogramm »Rechteck« mit einem Rechteck umrandet. Anschließend wartet das Programm auf den Mausklick des Anwenders. Ist dieser erfolgt, so wird mit MOUSE(3) die Position geholt, in die neue *farbe* umgerechnet und mit *SetAPen* gesetzt. Sobald die Maustaste losgelassen ist, wird die Schleife bei »warte« verlassen und zu »PointerPruefen« gesprungen.

Die Subroutine »markieren« sorgt für die Darstellung von Linien und Rechtecken während des Zeichenvorganges. Der Zeichen-Modus wird mit *SetDrMd* auf JAM1+COMPLEMENT (Wert 3) gesetzt. Durch COMPLEMENT werden die Linien komplementiert (invers) dargestellt. In der Routine wird durch das zweimalige Zeichnen jeweils der normale Zustand hergestellt. Die Linien werden also über dem Hintergrund gezeichnet, ohne ihn zu zerstören.

Dank der guten Vorbereitung bei »menukontr« und »mauskontr« und mit Hilfe kleinerer Unterprogramme fallen die einzelnen Zeichenroutinen recht kurz aus. Ein paar Zeiger sorgen dafür, daß nichts durcheinander kommt. Die jeweiligen Routinen haben wir bereits besprochen. Weitere Erklärungen erübrigen sich daher.

Kapitel 7

Transformationen von Liniengrafiken

Bereits im ersten Kapitel haben Sie erfahren, daß sich jeder Punkt des Bildschirmes mit einer X- und Y-Koordinate darstellen läßt. Für die Manipulation von grafischen Objekten und Bildschirmausschnitten reicht diese einfache Festlegung nicht aus. Stellen Sie sich einmal das Durcheinander bei der Berechnung vor, wenn Sie mehrere Liniengrafiken auf dem Bildschirm rotieren lassen. Aber keine Bange, mit einer simplen Technik werden wir die Angelegenheit schnell in den Griff bekommen.

Daß Grafik immer auch Mathematik bedeutet, haben Sie inzwischen schon erfahren. Je nachdem, wie sich Ihr Verhältnis zur Mathematik entwickelt hat, werden Sie diesen Umstand bedauern oder erfreut zur Kenntnis genommen haben. Sollten Sie zu dem statistisch gesehen größeren Teil der Bevölkerung gehören, der der Zahlen- und Buchstabenrechnung nicht viel abgewinnen kann, so darf ich Sie beruhigen. Die Berechnungen in diesem Abschnitt sind sehr einfach gehalten, damit Sie vor lauter theoretischem Frust nicht die Lust am Manipulieren der Grafiken verlieren. Mehr als die Kenntnisse der Winkelfunktionen werden nicht benötigt. Ein kleines Kapitel über dieses Thema wird etwaige Erinnerungslücken schnell wieder füllen.

Natürlich gibt es immer verschiedene Wege, um etwas zu berechnen oder zu programmieren. Zur Programmierung der Transformationen habe ich den meiner Meinung nach einfachsten gewählt. Schließlich muß das Programmieren nicht unbedingt in stumpfsinnige Arbeit ausarten. Wenn Sie sich jedoch auf die Liniengrafik spezialisieren wollen, weil Sie diese zum Beispiel beruflich benötigen, so empfehle ich Ihnen, sich weiterführende Literatur zu diesem Thema zu beschaffen. Nun aber zur einfachen Darstellung. Es beginnt sogar sehr einfach. Allerdings wird es im Laufe des Kapitels schon ein klein wenig komplizierter.

7.1 Objekte im Koordinatenkreuz

Der Grundgedanke bei der Transformation (Umwandlung) von Grafiken ist, das zu manipulierende Objekt von den Koordinaten des Bildschirmes zu lösen. Dazu verpassen wir dem Objekt einfach ein eigenes Koordinatenkreuz innerhalb des Koordinatensystems des Bildschirmes. Bei zwei Objekten hätten wir dann zwei zusätzliche Koordinatenkreuze innerhalb der Bildschirmkoordinaten usw. Bei den folgenden Beispielen werden wir immer nur ein Objekt und eine Art der Transformation behandeln. Die Kombination von mehreren Objekten mit unterschiedlichen Manipulationen dürfte am Ende des Abschnittes für Sie kein Problem mehr darstellen.

Mit diesem eigenen Koodinatensystem können wir nun schalten und walten, wie wir wollen. Was außerhalb der Objekt-Koordinaten geschieht, wird vorerst einfach ignoriert. Erst wenn es erforderlich wird, werden die X- und Y-Abstände dieses separaten Koordinatensystems zu den ermittelten Werten der Transformation hinzugefügt. Damit lassen sich die Grafiken selbst natürlich wesentlich einfacher berechnen.

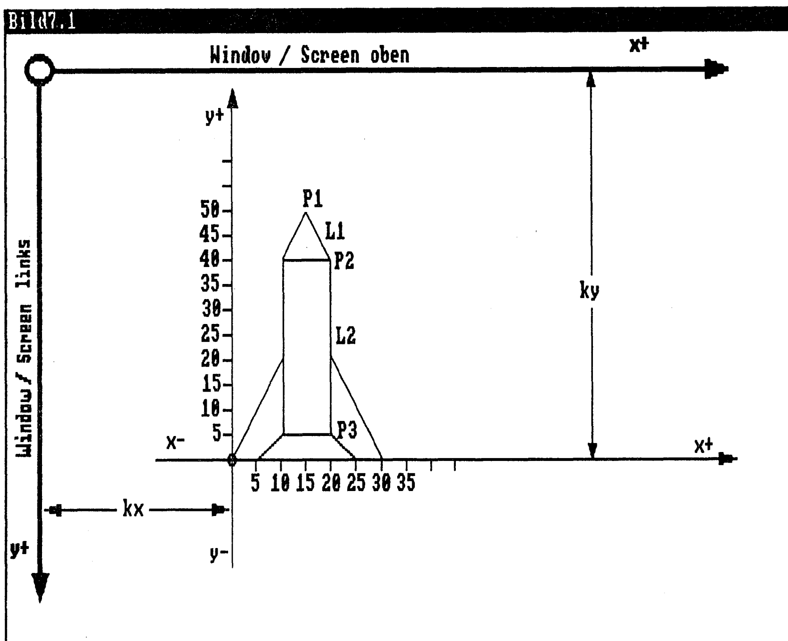


Bild 7.1: Ein Objekt mit eigenem Koordinatensystem

In Bild 7.1 ist das Prinzip dargestellt. In einem Koordinatensystem ist im Schnittpunkt der X-Achse mit der Y-Achse der Nullpunkt. Der nach rechts abgehende Strahl der X-Achse ist positiv und links vom Nullpunkt ist er negativ. Bei der Y-Achse zeigt der nach

oben gehende Strahl positive Werte und der Pfeil, der vom Nullpunkt abwärts gerichtet ist, enthält die negativen Zahlen. Diese Grundregel würde natürlich auch für das Koordinatensystem des Bildschirms gelten. Daß das beim Computer nicht berücksichtigt wurde, habe ich bereits zu Beginn des Buches erwähnt. Da beim Computer der Nullpunkt in der linken oberen Ecke sitzt, müßten die Y-Werte eigentlich negativ sein.

Kommen wir nun zu unserem Objekt, der Rakete aus Bild 7.1, zurück. Die Zeichnung setzt sich aus 11 Linien zusammen. Jede dieser Linien wird durch zwei Punkte definiert. Jeder Punkt wiederum ist durch seine Koodinaten x und y festgelegt. Für jede Linie benötigen wir also vier Koordinatenwerte. In unseren Beispielen nennen wir sie StartpunktX (sx), StartpunktY (sy), EndpunktX (ex) und EndpunktY (ey).

Linie n: Punkt n(sx,sy) und Punkt n+1(ex,ey)
Linie L1: P1(15,50) und P2(20,40)
Linie L2: P2(20,40) und P3(20,5)

Zur Speicherung des gesamten Objektes bietet sich die Ablage in vier eindimensionalen Feldern für jeden der vier Koordinatenwerte einer Linie an. Die Anzahl der Feldelemente entspricht dann der Anzahl der Linien des Objektes.

Linie n: (sx(n),sy(n)) und (ex(n),ey(n))
Linie L1: (sx(0),sy(0)) und (ex(0),ey(0))
Linie L2: (sx(1),sy(1)) und (ex(1),ey(1))

Damit haben wir bereits den Befehl zum Zeichnen einer Linie und durch die Anzahl der Linien die Schleife zur Erstellung des gesamten Objektes fertig:

```
FOR Linie = Ø to n
  LINE (sx(Linie),sy(Linie))-(ex(Linie),ey(Linie))
NEXT
```

Sie sehen, bisher war nichts Kompliziertes dabei. Natürlich werden diese Felder, je nach der Art der Transformation, verändert. Dazu kommen wir anschließend in den einzelnen Kapiteln dieses Abschnittes. Um das Objekt richtig auf den Bildschirm zu bringen, brauchen wir nur noch den Abstand der Objekt-Koordinaten *kx* und *ky* zu den Bildschirm-Koordinaten hinzuzufügen:

```
FOR Linie = Ø to n
  LINE (sx(Linie)+kx,sy(Linie)+ky)-(ex(Linie)+kx,ey(Linie)+ky)
NEXT
```

Damit haben wir die erste Hürde bereits genommen.

7.2 Objekt-Animation

Beginnen wir mit der einfachsten Transformation, mit der Verschiebung von Grafiken auf dem Bildschirm. Die Bewegung von Objekten auf dem Screen dürfte auch die am häufigsten angewendete Technik dieser Art sein. Für viele Anwendungsgebiete kann sie den Einsatz von BOBs und Sprites ersetzen, da sie recht unkompliziert einsetzbar ist.

7.2.1 Bewegen von Liniengrafiken

Zum einfacheren Verständnis des Prinzips betrachten wir uns am besten die Verschiebung einer Liniengrafik an einer kleinen Skizze:

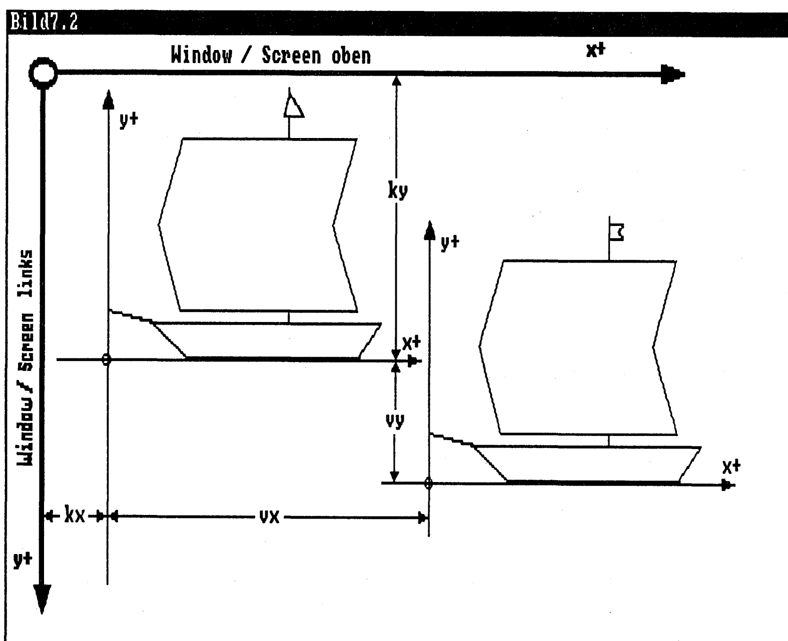


Bild 7.2: Verschieben von Liniengrafiken

Die beiden Koordinatenkreuze innerhalb der Bildschirmkoordinaten zeigen die Grafik vor und nach der Verschiebung. Der Abstand der beiden Koordinatensysteme zueinander in X-Richtung wird als *vx* und in Y-Richtung als *vy* bezeichnet. Wie nicht anders zu erwarten, hat sich innerhalb der Koordinatensysteme keine Veränderung ergeben. Die einfache rechnerische Darstellung der Verschiebung stellt sich wie folgt dar:

```
kx = kx + vx
ky = ky + vy
```

Daß man die Verschiebung, auch ohne eigenes Koordinatensystem für das zu verschiebende Objekt, wesentlich einfacher programmieren kann, ist keine Frage. Aber wie bereits erwähnt, gilt das nicht für die komplizierteren Transformationen wie die Spiegelung oder die Drehung von Grafiken. Wir werden also den einmal eingeschlagenen Weg beibehalten.

7.2.2 Viele Wege führen nach Rom

Zum Verschieben einer Grafik stellt uns der Amiga eine Reihe von Techniken zur Auswahl. Allerdings ist nur eine davon für unseren Weg geeignet. Schauen wir zuerst, welche Möglichkeiten wir anwenden können.

Die SCROLL-Anweisung kann zwar einen rechteckigen Bildschirmausschnitt in jede Richtung bewegen. Jedoch geschieht die Verschiebung innerhalb eines größeren Rechteckes. Dabei wird der Hintergrund der Grafik zerstört. Das Wichtigste jedoch ist, daß die einzelnen Koordinaten der Linien nicht erfaßt werden. Daher ist die SCROLL-Anweisung für die rechnerische Grafik-Transformation nicht geeignet.

Die zweite Möglichkeit der Grafik-Verschiebung bildet das Anweisungspaar GET und PUT. Durch den richtigen Modus kann eine Zerstörung des Hintergrundes vermieden werden. Aber auch hier wird die komplette Grafik bzw. ein rechteckiger Bildschirmbereich verschoben. Daher sind diese beiden, sehr effektvoll zu programmierenden Grafikbefehle für die Liniengrafik ungeeignet.

Eine weitere Alternative bildet die einfache LINE-Anweisung. Wir hatten sie bereits kurz angesprochen. Durch die Wahl des richtigen Zeichenmodus kann die Zerstörung des Hintergrundes vermieden werden. Die Start- und die Endkoordinaten können gespeichert und weiterverarbeitet werden. Damit haben wir alle Voraussetzungen erfüllt. Ein kleines Beispiel wird uns das demonstrieren:

```
FOR i=Ø TO 1Ø:sx(i)=i+5:ex(i)=i*2Ø:sy(i)=i*7:ey(i)=i+8Ø:NEXT
kx=1Ø:ky=2Ø:vx=4:vy=1:FOR n=Ø TO 5Ø:FOR i=Ø TO 1Ø
LINE(sx(i)+kx,sy(i)+ky)-(ex(i)+kx,ey(i)+ky),Ø
NEXT i: kx=kx+vx:ky=ky+vy:FOR i=Ø TO 1Ø
LINE(sx(i)+kx,sy(i)+ky)-(ex(i)+kx,ey(i)+ky),Ø:NEXT i,n
```

Anstelle der LINE-Anweisung können Sie natürlich auch die entsprechenden Library-Routinen einsetzen. Das Teilprogramm, das so nicht lauffähig ist, könnte dabei folgendes Aussehen haben:

```
FOR n= Ø TO i
  CALL Move&(rp&,sx(n)+kx,sy(n)+ky)
  CALL Draw&(rp&,ex(n)+kx,ey(n)+ky)
NEXT
```

Nachdem wir nun wissen, wie es geht, können wir unser Wissen in einem praktischen Beispiel einsetzen.

7.2.3 Follow me

Das folgende Programm enthält ein Mini-Zeichenprogramm, mit dem Sie jede beliebige Liniengrafik zeichnen können. Ein karierter Untergrund erleichtert das Zeichnen. Anschließend folgt die Grafik jeder Mausbewegung.

```
REM FollowMe  Pfad: Darstellung/7TransLinie/FollowMe
'P7-1
'sx() Startpunkt Linie X-Achse
'ex() Endpunkt Linie X-Achse
'sy() Startpunkt Linie Y-Achse
'ey() Endpunkt Linie Y-Achse
'kx   Koordinatenkreuz des Objektes X-Achse
'ky   Koordinatenkreuz des Objektes Y-Achse
'vx   Verschiebung X-Achse
'vy   Verschiebung Y-Achse
'Verschiebung X-Achse=sx()+vx  und ex()+vx
'Verschiebung Y-Achse=sy()+vy  und ey()+vy
'
DEFINT a-z
d=4Ø        'Anzahl der Linien zum Zeichnen
DIM sx(d),ex(d),sy(d),ey(d)
SCREEN 1,32Ø,256,2,1
WINDOW 2,"follow me",,Ø,1
win2&=WINDOW(7)
PALETTE 2,Ø,Ø,.8
start:
CLS:Netz
LOCATE 1,1:PRINT "bitte Objekt zeichnen"
LOCATE 2,1:PRINT "RETURN = fertig"

i=-1
zeichnen:
```

```
i=i+1:muss=MOUSE(Ø)
WHILE MOUSE(Ø)=Ø
  ta-=INKEY-:IF ta-=CHR-(13)  THEN i=i-1:GOTO Koordinaten
WEND
muss=MOUSE(Ø):sx(i)=MOUSE(3):sy(i)=MOUSE(4)
WHILE MOUSE(Ø)<>Ø :WEND
ex(i)=MOUSE(5):ey(i)=MOUSE(6)
LINE (sx(i),sy(i))-(ex(i),ey(i))
IF i>=d THEN Koordinaten
IF ta-<>CHR-(13) THEN zeichnen

Koordinaten:
kx=WINDOW(2)         'Maximum fuer x-Koordinate
ky=Ø                 'Minimum fuer y-Koordinate
FOR n=Ø TO i
  IF sx(n)<kx THEN kx=sx(n)
  IF ex(n)<kx THEN kx=sx(n)
  IF sy(n)>ky1 THEN ky=sy(n)
  IF ey(n)>ky1 THEN ky=ey(n)
NEXT
FOR n=Ø TO i
  sx(n)=sx(n)-kx
  ex(n)=ex(n)-kx
  sy(n)=sy(n)-ky
  ey(n)=ey(n)-ky
NEXT
CLS:COLOR 3

verschieben:
LOCATE 1,1:PRINT "Das Objekt folgt jeder Mausbewegung"
LOCATE 2,1:PRINT "RETURN = Neustart    Taste = fertig"
x1=PEEKW(win2&+14):y1=PEEKW(win2&+12):COLOR 1

aktion:
x2=PEEKW(win2&+14):y2=PEEKW(win2&+12)
vx=x2-x1 :vy=y2-y1              'Verschiebungsvektoren
ta-=INKEY-: IF ta-<>"" THEN ende
IF vx=Ø AND vy=Ø THEN aktion   'keine Mausbewegung
FOR n=Ø TO i
  LINE (sx(n)+kx,sy(n)+ky)-(ex(n)+kx,ey(n)+ky),Ø
NEXT
kx=kx+vx:ky=ky+vy
```

```
FOR n=Ø TO i
  LINE (sx(n)+kx,sy(n)+ky)-(ex(n)+kx,ey(n)+ky),1
NEXT
x1=x2:y1=y2
IF ta-="" THEN aktion
ende:
IF ta-=CHR-(13) THEN start
WINDOW CLOSE 2: SCREEN CLOSE 1
LIBRARY CLOSE
ERASE sx,ex,sy,ey
END

SUB Netz STATIC
FOR y = Ø TO WINDOW(3) STEP 5
  LINE (Ø,y)-(WINDOW(2),y),2
NEXT
FOR x = Ø TO WINDOW(2) STEP 5
  LINE (x,Ø)-(x,WINDOW(3)),2
NEXT
END SUB
```

Da dieses das erste Programm aus einer Reihe von Programmen über die Manipulation von Grafiken ist, sind zu Programmbeginn die wichtigsten Variablen aufgelistet. Das Programm beginnt mit der Dimensionierung der Start- und Endkoordinaten der Linien. Damit lassen sich grafische Objekte zeichnen, die aus maximal 41 Linien zusammengesetzt sind. Haben Sie das Bedürfnis, detailliertere Grafiken für die Verschiebung zu zeichnen, so ändern Sie einfach die Dimensionierung.

Es wird ein neuer Bildschirm mit niedriger Auflösung und ein bildschirmfüllendes Fenster geöffnet. Die PALETTE-Anweisung setzt die Farbe 2 auf eine etwas hellere Farbe, als die normale Hintergrundfarbe zeigt. Damit wird in dem Unterprogramm »Netz« ein karierter Untergrund ausgegeben, der das Zeichnen der Grafik erleichtern soll.

Beim Label »zeichnen« können Sie dann Ihre Zeichenkünste auf den Bildschirm zaubern. Die Variable *i* zählt dabei die Anzahl der gezeichneten Linien für die vier Feldvariablen. Die Startpunkte der Linien werden aus MOUSE(3) und MOUSE(4) geholt und in den Variablen *sx()* und *sy()* festgehalten. Die Funktionen MOUSE(5) und MOUSE(6) liefern dagegen die Endpunkte der Linien, für die die Variablen *ex()* und *ey()* zuständig sind. Mit einer einfachen LINE-Anweisung werden die Linien schließlich gezeichnet. Hat der Anwender RETURN gedrückt, oder ist die maximale Anzahl der Linien erreicht, so ist das Kunstwerk vollendet, und es wird zur Sprungmarke »Koordinaten« verzweigt.

Wie schon der Name sagt, werden bei »Koordinaten« die Koordinaten des Objektes errechnet. Für die X-Achse werden dazu sämtliche Start- und Endkoordinaten *sx* und *ex* miteinander verglichen, bis der kleinste Wert ermittelt ist. Dieser kleinste Wert wird als Koordinatenabstand-X *kx* festgelegt. Auf die gleiche Weise wird der Koordinatenabstand-Y *ky* errechnet. In einer zweiten FOR/NEXT-Schleife werden dann alle Koordinatenwerte um *kx* und *ky* verringert. Nachdem nun das Objekt sein eigenes Koordinatensystem hat, wird es mit CLS vom Bildschirm gelöscht.

Bei »verschieben« werden aus der Window-Struktur die Positionswerte des Mauszeigers geholt. Die X-Koordinate finden wir in Wortlänge in der 14. Speicherstelle und die Y-Koordinate bei einem Offset von 12 der Startadresse des Fensters. Beim Label »aktion« geht's dann richtig los! Wieder werden die Koordinaten des Mauspfeiles geholt. Ergibt sich eine Veränderung zu den Variablen *x1* und *y1*, so wurde die Maus bewegt. Der zurückgelegte Weg wird an die Variablen *vx* und *vy* übergeben. Die Grafik der alten Position wird nochmals mit der Hintergrundfarbe gezeichnet, so daß sie auf Nimmerwiedersehn verschwindet. Nun werden die Koordinatenwerte *kx* und *ky* um die Verschiebung(en) um *vx* und *vy* erhöht (oder verringert). Mit den neuen Werten wird anschließend die Grafik neu gezeichnet. Die Verschiebung ist perfekt. Die aktuellen Mauspositionen werden zu alten Positionen (x1=x2:y1=y2) und der nächste Durchgang kann beginnen.

Durch einen Druck auf die RETURN-Taste wird das Programm neu begonnen und es kann eine neue Zeichnung erstellt werden, die dann ebenfalls verschoben werden kann. Ein Druck auf irgendeine andere Taste beendet das Programm mit den üblichen Aufräumungsarbeiten.

7.3 Liniengrafik im Spiegelbild

Jetzt wird es wieder interessant. Die Spiegelung von Grafiken bietet einige reizvolle Aspekte. Die meisten Zeichenprogramme bieten deshalb die Spiegelung von Teilgrafiken als Option an. Aber auch im technischen Bereich finden sich durchaus Einsatzgebiete. Ich denke dabei zum Beispiel an symmetrische Werkstücke, die durch Achsenspiegelung wesentlich schneller erstellt werden können.

7.3.1 Wie funktioniert die Spiegelung?

Bevor wir tiefer in dieses Thema einsteigen, schauen Sie sich bitte zuerst das Prinzip in der folgenden Skizze an:

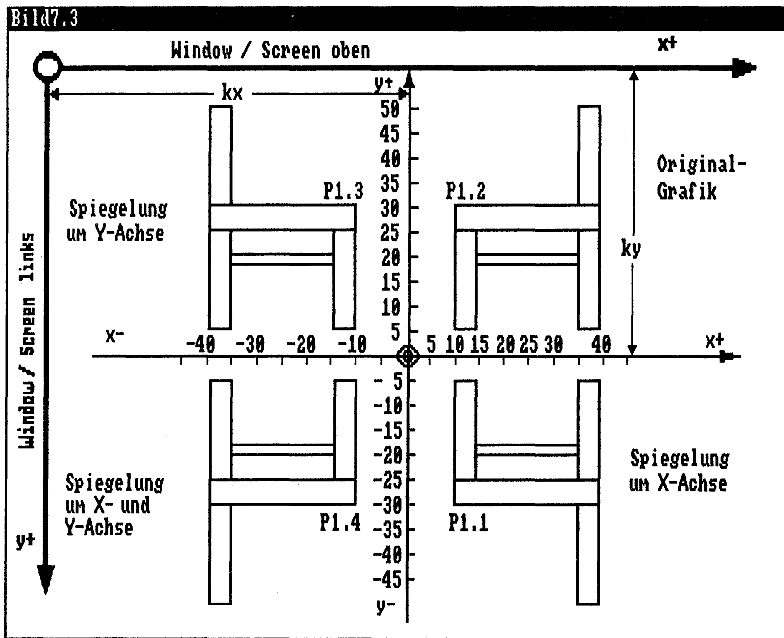


Bild 7.3: Spiegelung einer Liniengrafik um eine oder zwei Achsen

Im oberen rechten Viertel des Koordinatenkreuzes befindet sich das Objekt, das gespiegelt werden soll. Der Einfachheit halber betrachten wir nur einen Punkt der Grafik, der in der Skizze P1 genannt ist. Das Ergebnis unserer Betrachtungen läßt sich bei der Spiegelung auf alle Punkte der Grafik übertragen.

Nehmen wir uns zuerst die Spiegelung an der X-Achse zur Brust. Unser Referenzpunkt P1 wird dabei auf die Position P1.1 gespiegelt. Wie bei jeder Transformation interessiert uns dabei die Veränderung der X- und Y-Koordinaten-Werte:

P1: x=+10 y=+30
P1.1: x=+10 y=– 30

Es hat sich bei der Spiegelung an der X-Achse lediglich das Vorzeichen des Y-Wertes verändert.

Als zweites betrachten wir die Spiegelung an der Y-Achse. Der Punkt P1.2 stellt den gespiegelten Referenzpunkt P1 dar. Die X- und Y-Koordinaten zeigen dabei folgende Veränderung:

P1: x=+10 y=+30
P1.2: x=– 10 y=+30

Bei der Spiegelung an der Y-Achse hat sich nur das Vorzeichen des X-Wertes verändert.

Die letzte Spiegelung stellt eine Doppelspiegelung dar. Die Grafik wird also gleichzeitig an der X- und Y-Achse gespiegelt. Unser Referenzpunkt P1 erscheint nun als Punkt P1.3. Wie haben sich die X- und Y-Koordinaten-Werte dieses Mal verändert?

P1: x=+10 y=+30
P1.3: x=− 10 y=− 30

Die Doppelspiegelung an den beiden Achsen verändert also die Vorzeichen beider Koordinaten-Werte. Fassen wir das Ergebnis unserer Betrachtungen in einer kleinen Tabelle zusammen:

Spiegelung	**X-Wert**	**Y-Wert**
um die X-Achse	bleibt	* −1
um die Y-Achse	* −1	bleibt
um die X- und Y-Achse	* −1	* −1

Damit genug der Theorie. Bleibt uns nur noch eines, die Erkenntnisse in einem praktischen Beispiel auszuprobieren:

7.3.2 Line Mirror

Zuerst einige Worte zum Programmablauf. Sie können wieder Ihre zeichnerischen Talente in einem kleinen Zeichenprogramm ausprobieren. Eine vollständige Spiegelung erhalten Sie, wenn Sie Ihre Grafik in das obere rechte Viertel des Bildschirms zeichnen. An einer anderen Stelle könnte das gespiegelte Objekt (je nach Größe) nur unvollständig zu sehen sein. Einen Programmabbruch bei Überschreitung der Bildschirmgrenzen brauchen Sie trotz der zum Zeichnen verwendeten Bibliotheks-Routinen nicht zu befürchten. Für die Anzahl der Linien gilt das gleiche, wie im ersten Programm des Abschnittes besprochen. Durch Betätigung der linken Maus-Taste erscheint ein Requester. In ihm können Sie anklicken, ob Sie eine bestimmte Spiegelung sehen wollen oder nicht. Auch das Ende des Programmes oder einen Neustart wählen Sie durch einen Requester.

```
REM LineMirror  Pfad: Darstellung/7TransLinie/LineMirror
'P7-2
'sx() Startpunkt Linie X-Achse
'ex() Endpunkt Linie X-Achse
'sy() Startpunkt Linie Y-Achse
'ey() Endpunkt Linie Y-Achse
'kx   Koordinatenkreuz des Objektes X-Achse
```

```
'ky   Koordinatenkreuz des Objektes Y-Achse
'ssx() Startpunkt Linie X-Achse der Spiegelung
'sex() Endpunkt Linie X-Achse der Spiegelung
'
CLEAR
DEFINT a-z
d=4Ø       'Anzahl der Linien zum Zeichnen
DIM sx(d),ex(d),sy(d),ey(d),ssx(d),sex(d),ssy(d),sey(d)
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION AutoRequest&() LIBRARY
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"

art&=3+(2^16)
rek&=Ø:rk&=VARPTR(rek&)
m1&=AllocRemember&(rk&,1ØØ,art&)
IF m1&=Ø THEN PRINT "memory-FEHLER":BEEP:GOTO ende

txt3-="ja"
txt4-="nein"

SCREEN 1,32Ø,256,2,1
WINDOW 2,"LineMirror",,Ø,1
win2&=WINDOW(7)
rp2&=WINDOW(8)

start:
CLS
LOCATE 1,1:PRINT "bitte Objekt zeichnen"
LOCATE 2,1:PRINT "RETURN = fertig"
i=-1
zeichnen:
i=i+1:muss=MOUSE(Ø)
WHILE MOUSE(Ø)=Ø
  ta-=INKEY-:IF ta-=CHR-(13)  THEN i=i-1:GOTO Koordinaten
WEND
muss=MOUSE(Ø):sx(i)=MOUSE(3):sy(i)=MOUSE(4):PSET (sx(i),sy(i))
WHILE MOUSE(Ø)<>Ø :WEND
ex(i)=MOUSE(5):ey(i)=MOUSE(6)
LINE (sx(i),sy(i))-(ex(i),ey(i))
IF i>=4Ø THEN Koordinaten
IF ta-<>CHR-(13) THEN zeichnen
```

```
Koordinaten:
kx=WINDOW(2)         'Maximum fuer x-Koordinate
fb=kx
fh=WINDOW(3)         'Fensterhoehe
ky=Ø               'Minimum fuer y-Koordinate
FOR n=Ø TO i
  IF sx(n)<kx THEN kx=sx(n)
  IF ex(n)<kx THEN kx=ex(n)
  IF sy(n)>ky THEN ky=sy(n)
  IF ey(n)>ky THEN ky=ey(n)
NEXT
FOR n=Ø TO i
  sx(n)=sx(n)-kx
  ex(n)=ex(n)-kx
  sy(n)=sy(n)-ky
  ey(n)=ey(n)-ky
NEXT

spiegeln:
LOCATE 1,1:PRINT SPACE-(21)
LOCATE 2,1:PRINT SPACE-(15)
txt1-="Soll das Objekt um die "
txt2-="X-Achse gespiegelt werden?"
GOSUB Requester
IF jn& THEN GOSUB spiegelnX
txt1-="Soll das Objekt um die "
txt2-="Y-Achse gespiegelt werden?"
GOSUB Requester
IF jn& THEN GOSUB spiegelnY
txt1-="Soll das Objekt um beide "
txt2-="Achsen gespiegelt werden?"
GOSUB Requester
IF jn& THEN GOSUB spiegelnXY
txt1-="Wollen Sie einen weiteren"
txt2-="Versuch wagen?"
GOSUB Requester
IF jn& THEN start

ende:
WINDOW CLOSE 2
SCREEN CLOSE 1
CALL FreeRemember&(rk&,-1)
```

```
LIBRARY CLOSE
ERASE sx,ex,sy,ey,ssx,sex,ssy,sey
END

spiegelnX:
FOR n=Ø TO i
  ssx(n)=sx(n)
  sex(n)=ex(n)
  ssy(n)=-sy(n)
  sey(n)=-ey(n)
NEXT
GOTO aktion

spiegelnY:
FOR n=Ø TO i
  ssx(n)=-sx(n)
  sex(n)=-ex(n)
  ssy(n)=sy(n)
  sey(n)=ey(n)
NEXT
GOTO aktion

spiegelnXY:
FOR n=Ø TO i
  ssx(n)=-sx(n)
  sex(n)=-ex(n)
  ssy(n)=-sy(n)
  sey(n)=-ey(n)
NEXT

aktion:
CLS:COLOR 3
CALL Move&(rp2&,kx,Ø)
CALL Draw&(rp2&,kx,fh)
CALL Move&(rp2&,Ø,ky)
CALL Draw&(rp2&,fb,ky)
COLOR 1
FOR n=Ø TO i
  CALL Move&(rp2&,sx(n)+kx,sy(n)+ky)
  CALL Draw&(rp2&,ex(n)+kx,ey(n)+ky)
NEXT
FOR n=Ø TO i
  CALL Move&(rp2&,ssx(n)+kx,ssy(n)+ky)
```

```
  CALL Draw&(rp2&,sex(n)+kx,sey(n)+ky)
NEXT
warten:
muss=MOUSE(Ø):WHILE MOUSE(Ø)=Ø :WEND
RETURN

Requester:
CALL Texte (txt1-,Ø,Ø,m1&+2Ø)
CALL Texte (txt2-,1Ø,2Ø,n&)
CALL Texte (txt3-,Ø,4Ø,n&)
CALL Texte (txt4-,Ø,6Ø,n&)
jn&= AutoRequest&(WINDOW(7),m1&,m1&+4Ø,m1&+6Ø,Ø,Ø,25Ø,6Ø)
RETURN

SUB Texte (text-,y%,o%,n&) STATIC
SHARED m1&
text-=text-+CHR-(Ø)
StrukturIText:
POKE m1&+o%,1                'FrontPen
POKE m1&+o%+1,Ø              'BackPen
POKE m1&+o%+2,2              'DrawMode
POKEW m1&+o%+4,5             'LeftEdge
POKEW m1&+o%+6,5+y%          'TopEdge
POKEL m1&+o%+8,Ø             'TextAttr
POKEL m1&+o%+12,SADD(text-)  'text
POKEL m1&+o%+16,n&           'NextText
END SUB
```

Im Programm finden Sie zuerst die Dimensionierung der Start- und Endkoordinaten für 41 Linien. Anschließend werden zwei Bibliotheks-Routinen als aufrufbare Funktionen deklariert. Sie werden benötigt, um durch den Einsatz von Requestern den Programmkomfort zu erhöhen und dem Programm einen professionellen Anstrich zu verleihen. Die den Öffnungsroutinen folgende Speicherbelegung ist für die Text-Strukturen des Requesters.

Das Fenster und der Screen werden wie bereits bekannt geöffnet. Beim Label »zeichnen« werden wieder die Start- und End-Koordinaten der einzelnen Linien durch die MOUSE-Funktionen ermittelt und in den Feldvariablen gespeichert. Auch bei der Festlegung der Koordinaten wurde nichts verändert.

Interessant wird es wieder bei der Sprungmarke »spiegeln«. Mit den Stringvariablen *txt1$* und *txt2$* für den Requester-Text verzweigt das Programm zur Subroutine »Requester«. Schauen wir einmal, was dort passiert. Viermal wird das Unterprogramm »Texte«

aufgerufen. Jeder Unterprogramm-Aufruf legt eine Text-Struktur für den Requester fest. Er sorgt auch für die benötigte Textverknüpfung und positioniert den Text im Requester. Mit

```
jn&= AutoRequest&
```

wird der Requester aktiviert. Mit dem Wunsch des Anwenders, festgehalten in der Variablen *jn&*, springt das Programm zurück. Ist *jn&* logisch wahr, verzweigt das Programm zur Abarbeitung der entsprechenden Spiegelung, im ersten Fall zu »spiegelnX«. Ansonsten wird der nächste Requester aufgerufen. Bei der letzten Verzweigung wird entweder zum Label »start« zurückgesprungen oder das Programm beendet. Dazu werden bei »ende« das Fenster und der Screen geschlossen, der belegte Speicherbereich freigegeben, die Biblotheken geschlossen und die Feldvariablen freigegeben.

Kommen wir nun zum Kern des Programmes. Bei den einzelnen Sprungmarken »spiegelnX«, »spiegelnY« und »spiegelnXY« werden die Start- und die End-Koordinaten der einzelnen Linien an die neuen Feld-Variablen *ssx(), sex(), ssy()* und *sey()* übergeben. Damit wird sichergestellt, daß für weitere Spiegelungen die ursprünglichen Feld-Variablen *sx()* etc. ihren Startwert behalten. Gleichzeitig werden den neuen Feldvariablen die der Spiegelung entsprechenden Vorzeichen mitgegeben. Die positiven oder negativen Vorzeichen entsprechen den Erkenntnissen, die wir am Anfang des Kapitels gewonnen haben.

Beim Label »aktion« schließlich wird die tatsächliche Spiegelung durchgeführt. Dazu wird zuerst der Bildschirm gelöscht und sodann ein rotes Koordinatenkreuz gezeichnet. In der ersten Schleife wird die ursprüngliche Liniengrafik neu gezeichnet. Die zweite FOR/NEXT-Schleife bringt schließlich das gespiegelte Objekt auf den Bildschirm. Natürlich wird nun wieder der Abstand des Objekt-Koordinatensystems zum Bildschirm in den Variablen *kx* und *ky* dazugezählt. Das Programm wartet in einer Schleife so lange, bis die Maustaste betätigt wird.

Sie sehen, daß die Spiegelung selbst nur einige Programmzeilen lang ist. Sie können sie also ohne großen Aufwand in Ihren eigenen Programmen einsetzen.

7.4 Vergrößern und Verkleinern

Nachdem bisher alles so schön geklappt hat, eilen wir mit Riesenschritten zu einer weiteren Variante der Umwandlung von Grafiken. Die Veränderung der Größe einer Grafik bietet einige reizvolle Aspekte. Einer der Gesichtspunkte ist die scheinbare Darstellung einer Bildtiefe durch eine sich nähernde oder entfernende Grafik. Inwieweit ein solcher Fahreffekt realisiert werden kann, hängt von der Geschwindigkeit der Grafikdarstellung des Rechners ab. In diesem Kapitel sehen Sie, zu was unser Grafikwunder Amiga mit seinem Amiga-Basic zu diesem Thema leisten kann.

7.4.1 Prinzip der Größenveränderung

Bisher haben wir bei den beiden untersuchten Transformationen das Koordinatenkreuz des Objektes verschoben oder die Vorzeichen der Linien verändert. Dieses Mal kommt ein Faktor mit ins Spiel. Die Veränderung der Größe wird durch einen Multiplikator hervorgerufen. Die folgende Skizze zeigt uns, wie das funktionieren soll:

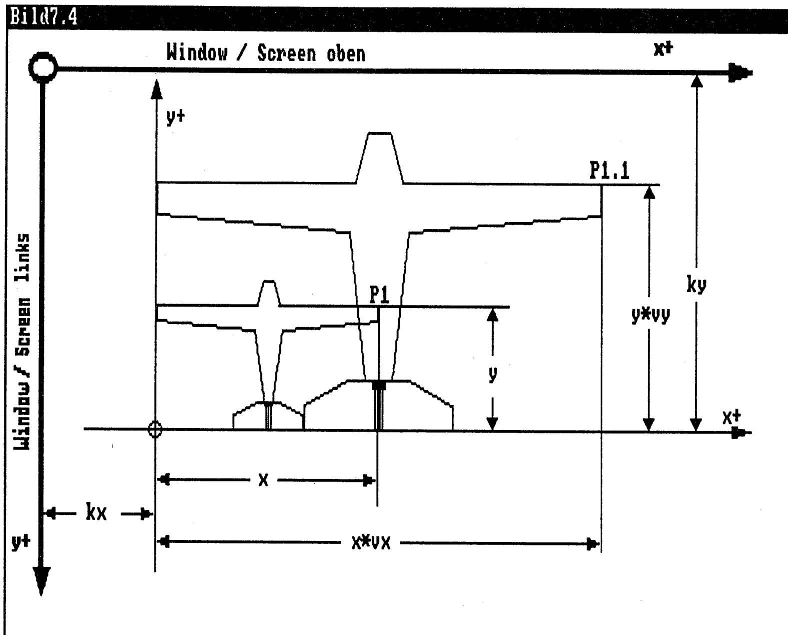


Bild 7.4: Vergrößern einer Liniengrafik

Das Bild 7.4 zeigt die gleichzeitige Vergrößerung eines Objektes um die X- und Y-Achse. Zur besseren Unterscheidung ist die ursprüngliche Grafik mit durchgezogenen Linien gezeichnet. Die vergrößerte Grafik dagegen ist mit gestrichelten Linien dargestellt. Betrachten wir die Veränderung der Referenzpunkte P1 und P1.1. Die Vergrößerung in X-Richtung erfolgt um den Faktor *vx*. Damit erhält der Punkt P1.1 den Koordinatenwert *x*vx*. Bei der Y-Achse verhält es sich ebenso. Den Vergrößerungsfaktor nennen wir *vy*. Dabei spielt es keine Rolle, ob das Objekt verkleinert oder vergrößert wird. Bei einer Vergrößerung ist der Faktor größer (>) als 1 und bei einer Verkleinerung kleiner (<) als 1.

vorher: P1(x,y)
vergrößert: P1.1(x*vx,y*vy)

Sie sehen, daß auch dieses Prinzip sehr einfach ist. Bevor wir die Theorie in die Tat umsetzen, wollen wir uns noch ein paar Gedanken über die Bildschirmdarstellung machen.

7.4.2 Double Buffering

Mit dem Programm *FollowMe* haben Sie selbstgezeichnete Liniengrafiken verschoben. Dabei ist ein recht unangenehmer Effekt aufgetreten. Während der Verschiebung mit der Maus war die Grafik verstümmelt auf dem Bildschirm zu sehen. Besonders störend machte sich das bei komplexen Grafiken, bestehend aus vielen Linien, bemerkbar. Die Ursache haben Sie sicher schnell erkannt. Auch beim Amiga dauert das Zeichnen von Linien eine gewisse Zeit, zumal wenn während des Zeichenvorganges noch Berechnungen durchgeführt werden. Durch das ständige Neuzeichnen der Grafik trat dann dieser unerwünschte Effekt zutage.

Dagegen müßte man doch etwas tun können. Selbstverständlich können und wollen wir etwas dagegen tun. Das Zauberwort dafür heißt »Double Buffering«. Man verlegt die Konstruktion der Grafik einfach auf einen oder mehrere unsichtbaren Speicherbereiche, zum Beispiel auf einen eigenen Screen oder in eine eigene Bitmap. Ist das Bild fertiggestellt, wird es einfach in die aktuelle Bitmap, also in das sichtbare Bild, kopiert.

Diese Technik werden wir bereits beim folgenden Programm anwenden. Wir verwenden dazu zum Aufbau der Grafik einen eigenen Screen. Sie werden staunen, um wieviel besser die Transformation auf dem Monitor wiedergegeben wird.

7.4.3 Kopieren mit Tempo und Komfort

Wie Sie bereits wissen, finden Sie normalerweise die Kurzbeschreibungen der verwendeten Library-Routinen im letzten Teil dieses Buches. Aus gutem Grund werden wir nun eine Ausnahme machen. Sie werden eine Routine kennenlernen, mit der wir wahre Wunderdinge vollbringen werden. Dabei kommen wir in Geschwindigkeitsbereiche, die kaum jemanden vermuten lassen, daß ein Basic-Programm dafür verantwortlich ist. Die Routine ist Bestandteil der Grafik-Bibliothek. Sehen wir uns zunächst die Syntax der Zauberformel an:

```
ClipBlit(Src,SrcX,SrcY,Dest,DestX,DestY,XSize,YSize,Minterm)
```

Die Routine verschiebt bzw. kopiert einen rechteckigen Bereich aus dem ursprünglichen RastPort an eine andere Position des Quell-RastPorts oder in einen anderen RastPort.

Noch genauer gesagt, ruft diese Routine zuerst eine andere Grafik-Routine *BltBitMap()* auf. Sie gilt aber nur für Verschiebungen innerhalb einer Bitmap bzw. auf eine andere BitMap. *ClipBlit* sorgt nun dafür, daß auch die anderen Grafik-Ausgabeelemente wie

die Layers etc. im zugehörigen Rastport richtiggestellt werden. Wenden Sie *BltBitMap* direkt in einem Intuition-Screen an, so kann ich Ihnen aus bitterer Erfahrung bestätigen, daß Sie dann einen fürchterlichen Speichersalat provozieren.

Die Erklärung der Library-Routine *ClipBlit* klingt so einfach, daß man auf Anhieb die Möglichkeiten, die dahinter stecken, glatt übersehen könnte. In dem folgenden Programm und in weiteren Programmen über die Transformation von Grafiken werden Sie eine Reihe von Anwendungen kennenlernen, die Ihnen die Leistungsfähigkeit der Routine vor Augen führt. Nehmen wir nun die einzelnen Parameter der Maschinen-Routine unter die Lupe.

`Src`
enthält einen Zeiger auf den RastPort, von dem aus der rechteckige Bereich kopiert wird. Man kann ihn auch als Quelle der Übertragung bezeichnen.

`SrcX, SrcY`
Die beiden Variablen enthalten die Koordinaten der oberen linken Ecke des zu übertragenden Rechteckbereiches des Quell-RastPorts.

`Dest`
Der Zeiger auf das Ziel des Kopiervorganges wird als *Dest* bezeichnet. Dieser Ziel-RastPort kann der gleiche RastPort der Quelle sein, oder irgendein anderer RastPort.

`DestX, DestY`
Die Position, oder genauer gesagt die Koordinaten der oberen linken Ecke des Rechteckes in dem Ziel-RastPort, auf den das Rechteck übertragen wird, werden durch diese beiden Parameter bestimmt.

`XSize, YSize`
Sie bestimmen die Größe des Rechteckes, welches kopiert wird. Dabei erhält *XSize* die Breite in Pixel und *YSize* die Höhe in Bildpunkten.

`Minterm`
Der letzte Parameter wird *Minterm* genannt. Das Wort ist eine Abkürzung für *minimum term* (minimale Bedingung). Während des Kopiervorganges können 256 (1 Byte) verschiedene Verknüpfungen, abhängig von den vier DMA-Kanälen, durchgeführt werden. Für uns sind dabei folgende Werte interessant:

`-3Ø` Das ursprüngliche Rechteck wird invertiert und dann kopiert.
`-5Ø` Das ursprüngliche Rechteck wird kopiert und am Ziel invertiert.
`-CØ` Es wird alles kopiert.

Wie bei vielen Library-Routinen müssen Sie auch bei *ClipBlit* eine gehörige Portion Vorsicht walten lassen. Da die Fehlererkennung des Amiga-Basic während der Ausführung der Routine außer Kraft gesetzt ist, müssen Sie beim Programmieren dafür sorgen, daß keine Fehler auftreten können. Zuerst sollten Sie tunlichst darauf achten, daß Sie nicht in Bereiche des Ziel-RastPorts kopieren, die gar nicht existieren bzw. außerhalb des RastPorts liegen. Das passiert sehr schnell, wenn die Werte von *XSize* oder *YSize* zu groß geraten sind. Damit überschreiben Sie einen unbekannten Speicherbereich. Meistens wird dieser Bereich gerade für irgendetwas Wichtiges benötigt. Die Folge ahnen Sie sicherlich. Richtig! Sie erhalten eine Grußmeldung des mächtigen Gurus.

Wenn Sie aber diese Vorsichtsmaßregeln beachten, kann Ihnen eigentlich nichts Unangenehmes mehr passieren. Sie werden die sagenhafte Kopiergeschwindigkeit der Routine *ClipBlit* nie mehr missen mögen.

7.4.4 Das Programm »Lupe«

Auch in diesem Programm wird Ihnen nicht irgendeine Liniengrafik vorgegeben. Sie sind wieder aufgefordert, eigene Kreationen auf den Bildschirm zu bringen. Besonders günstig für die Vergrößerung ist, wenn Sie in das linke untere Viertel des Bildschirmes zeichnen. Die Vergrößerung oder Verkleinerung wird über die vier Cursortasten gesteuert. Sobald die Zeichnung die Grenzen des Fensters verläßt, wird der Vergrößerungsvorgang mit einem Piepser abgebrochen. In einem solchen Fall sollten Sie den entgegengesetzten Pfeil drücken, um den Fehler rückgängig zu machen.

```
REM Lupe  Pfad: Darstellung/7TransLinie/Lupe
'P7-3
'sx() Startpunkt Linie X-Achse
'ex() Endpunkt Linie X-Achse
'sy() Startpunkt Linie Y-Achse
'ey() Endpunkt Linie Y-Achse
'vx   Vergroesserungsfaktor X-Achse
'vy   Vergroesserungsfaktor Y-Achse
'Vergroesserung X-Achse= sx()*vx und ex()*vx
'Vergroesserung Y-Achse= sx()*vy und ey()*vy
'das Programm arbeitet mit 2 Screens und 1 Window
'
IF FRE(-1)<80000& THEN BEEP:PRINT "Speicher zu klein":END
CLEAR
DEFINT a-z: d=40
DIM sx(d),ex(d),sy(d),ey(d)
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"
```

```
Video:
  SCREEN 1,32Ø,2ØØ,2,1
  WINDOW 2,,(Ø,Ø)-(31Ø,18Ø),Ø,1
  scr1&=PEEKL(WINDOW(7)+46)
  WINDOW CLOSE 2
  SCREEN 2,32Ø,2ØØ,2,1
  WINDOW 3,,(Ø,Ø)-(31Ø,18Ø),Ø,2
  scr2&=PEEKL(WINDOW(7)+46)
  rp1&=scr1&+84
  rp2&=scr2&+84
  PALETTE 2,Ø,Ø,.8

Netz
LOCATE 1,1:PRINT "bitte Objekt zeichnen"
LOCATE 2,1:PRINT "RETURN = fertig       "

i=-1
zeichnen:
  i=i+1
  WHILE MOUSE(Ø)=Ø
    ta-=INKEY-:IF ta-=CHR-(13)  THEN i=i-1:GOTO Koordinaten
  WEND
  muss=MOUSE(Ø):sx(i)=MOUSE(3):sy(i)=MOUSE(4)
  WHILE MOUSE(Ø)<>Ø :WEND
  ex(i)=MOUSE(5):ey(i)=MOUSE(6)
  LINE (sx(i),sy(i))-(ex(i),ey(i))
  IF i>=d THEN Koordinaten
  IF ta-<>CHR-(13) THEN zeichnen

Koordinaten:
  kx=WINDOW(2)          'Maximum fuer x-Koordinate
  ky=Ø                  'Minimum fuer y-Koordinate
  FOR n=Ø TO i
    IF sx(n)<kx THEN kx=sx(n)
    IF ex(n)<kx THEN kx=sx(n)
    IF sy(n)>ky THEN ky=sy(n)
    IF ey(n)>ky THEN ky=ey(n)
  NEXT
  FOR n=Ø TO i
    sx(n)=sx(n)-kx
    ex(n)=ex(n)-kx
    sy(n)=sy(n)-ky
    ey(n)=ey(n)-ky
  NEXT
```

```
fb=WINDOW(2):fh=WINDOW(3)
renderLeft=4:renderTop=2
skx=kx+renderLeft:sky=ky+renderTop 'Screenabmessung beachten
yanf=17:ymax=fh-yanf+renderTop

vergroessern:
  CLS:vx!=1:vy!=1
  LOCATE 1,1:PRINT "Cursortasten zum Vergroessern/Verkleinern"
  LOCATE 2,1:PRINT "Leertaste = ENDE"
  taste:ta-=INKEY-:IF ta-<CHR-(28) OR ta->CHR-(32) THEN taste
  wahl= ASC(ta-)-27
  ON wahl GOTO vergrY,verklY,vergrX,verklX,ende

vergrY:vy!=1.1*vy!:GOTO aktion
verklY:vy!= .9*vy!:GOTO aktion
vergrX:vx!=1.1*vx!:GOTO aktion
verklX:vx!= .9*vx!

aktion:
  clearw (rp1&)
  FOR n=Ø TO i
    asx=sx(n)*vx!+skx:asy=sy(n)*vy!+sky
    aex=ex(n)*vx!+skx:aey=ey(n)*vy!+sky
    IF asx>WINDOW(2) OR aex>WINDOW(2) THEN BEEP:GOTO taste
    IF asy<2Ø OR aey<2Ø THEN BEEP:GOTO taste
    CALL Move&(rp1&,asx,asy)
    CALL Draw&(rp1&,aex,aey)
  NEXT
  CALL ClipBlit&(rp1&,renderLeft,yanf,rp2&,renderLeft,yanf,
**                                              fb,ymax,192)
  GOTO taste

ende:
  WINDOW CLOSE 3 :SCREEN CLOSE 2 :SCREEN CLOSE 1
  LIBRARY CLOSE
  ERASE sx,sy,ex,ey
  END

SUB clearw (rp&) STATIC
  CALL SetRast&(rp&,Ø)
END SUB

SUB Netz STATIC
  FOR y = Ø TO WINDOW(3) STEP 5
```

```
    LINE (Ø,y)-(WINDOW(2),y),2
  NEXT
  FOR x = Ø TO WINDOW(2) STEP 5
    LINE (x,Ø)-(x,WINDOW(3)),2
  NEXT
END SUB
```

Nach der Variablen-Definition und den erforderlichen Dimensionierungen öffnen wir die Grafik- und die Intuition-Library. Anschließend öffnen wir zwei Screens. Der zweite Bildschirm wird für das Double Buffering benötigt. Um später direkt in die Bitmaps zeichnen zu können, brauchen wir die Adressen der der beiden RastPort-Strukturen. Diese finden wir an der Speicherstelle 84 der zugehörigen Screen-Struktur und halten sie in den Variablen *rp1&* und *rp2&* fest.

Der Anwender zeichnet nun seine Liniengrafik, wie bereits besprochen, in das Netz mit den Hilfslinien. Auch die Ermittlung der Koordinaten kennen Sie bereits. Es folgt die Festlegung der Variablen, die die unterschiedlichen Abmessungen zwischen Screen und Window berücksichtigen. Damit nähern wir uns bereits dem Kern des Programmes. Bei der Sprungmarke »vergroessern« werden die einzelnen Vergrößerungen oder Verkleinerungen der Liniengrafik eingeleitet.

Der Startwert der Vergrößerungsfaktoren *vx!* und *vy!* ist 1. Das bedeutet, das Bild wird in den gezeichneten Abmessungen auf den Bildschirm gebracht. Je nachdem, welche Cursortaste der Anwender nun betätigt, werden die beiden Vergrößerungsfaktoren *vx!* und *vy!* verändert. Bei jeder Verkleinerung werden die Faktoren mit 0,9 multipliziert und bei jeder Vergrößerung mit 1.1. Wenn Ihnen eine andere Schrittweite besser gefällt, so brauchen Sie nur diese Faktoren zu ändern. Das Programm verzweigt mit den neuen Werten zum wichtigsten Teil des Programmes.

Bei der Sprungmarke »aktion« kommt Bewegung in die Sache. Das ganz und gar nicht geheimnisvolle Double Buffering wird durchgeführt. Dazu wird zuerst in der Subroutine »clearw ()« der unsichtbare Screen Nummer 1 (rp1&) mit Hilfe der Library-Routine *SetRast&* mit der Hintergrundfarbe gelöscht. In der FOR/NEXT-Schleife wird nun die Grafik, für den Anwender unsichtbar, gezeichnet. Die neuen Start- und Endpunkte der einzelnen Linien werden mit dem aktuellen Faktor berechnet und an die Variablen *asx* (Startpunkt X) bis *aey* (Endpunkt Y) übergeben. Sie sehen, daß die neuen Koordinaten, bei jeder Größenveränderung, die Anfangskoordinaten als Rechengrundlage verwenden. Würden wir die geänderten Werte erneut ändern, so wäre bald kein vernünftiges Bild mehr zu sehen. Bei jeder Rechnung muß ja zum Setzen der Punkte auf- oder abgerundet werden. Nach einigen Tastendrücken hätten wir bereits irreguläre Werte. Sie können mit der beim Programm angewandten Methode verblüffende Effekte erzielen. Verkleinern Sie einmal Ihre Grafik bis zur Größe eines einzigen Pixels. Sie werden sehen, das ist überhaupt kein Problem. Stellen Sie sich einmal die Gesichter Ihrer Bekannten vor, wenn Sie aus einem einzelnen Punkt eine tolle Grafik hervorzaubern.

Zurück zu unserem Programm. Bevor die neuen Linien gezeichnet werden, erfolgt eine Überprüfung. Überschreiten die Werte die Fenstergrenzen (welche wir uns von dem sichtbaren Screen ausleihen), so wird zurückgesprungen. Andernfalls könnte, wie bereits erwähnt, beim nachfolgenden Zeichnen mit *Move&* und *Draw&* der Amiga abstürzen. Da der Faktor im Programm nicht zurückgeändert wird, sollten Sie in einem solchen Fall anschließend die entgegengesetzte Cursortaste drücken, damit das Programm wieder zeichenbare Werte zur Verfügung hat. Zum Schluß kopieren wir das neue Bild mit *ClipBlit&* in den sichtbaren Screen, bzw. in dessen Fenster.

Durch Betätigung der Leertaste kann das Programm beendet werden. Beim Label »ende« werden damit die üblichen Aufräumungsarbeiten ausgeführt. Haben Sie Geschmack an dieser Art der Grafik-Manipulation gefunden? Dann steht Ihren eigenen Aktivitäten nichts mehr im Wege.

7.5 Rotation

Die Rotation von Grafiken ist die optisch eindrucksvollste der Animationsarten. Durch eine gleichmäßige Drehung läßt sich ein natürlicher Vorgang am besten vortäuschen. Die Technik wird Ihnen nicht ganz unbekannt sein. In dem Kapitel über die Flächen haben wir damit bereits eine Uhr simuliert.

Nun werden wir auch, wie bereits eingangs erwähnt, die Winkelfunktionen einsetzen. Eine kleine Zusammenfassung der Formeln soll demjenigen helfen, dem die Begriffe Sinus oder Cosinus nicht oder nicht mehr geläufig sind. Damit können Sie bei Unklarheiten in diesen oder anderen Programmen jederzeit hier nachschlagen, ohne Ihr Mathematikbuch vom Speicher holen zu müssen.

In diesen Zusammenhang gehört auch die Wahl des Bildschirm-Modus. Haben Sie sich schon gewundert, warum alle Programme dieses Abschnittes einen Bildschirm mit niedriger Auflösung einsetzen? Nun, das liegt am Verhältnis der Grafikpunkte der X- zur Y-Achse. Dieses Thema haben wir bereits bei den Screens besprochen. Lores wurde gewählt, da dabei das Verhältnis X zu Y = 10 : 11 zum Tragen kommt. Die Punkte sind also nahezu quadratisch. Damit kann zur einfacheren Darstellung der Berechnungen ein Faktor, der das X/Y-Verhältnis berücksichtigt, entfallen. Bei den zurückliegenden Programmen der 2-D-Grafik hätten wir genausogut Bildschirme mit hoher Auflösung einsetzen können, da dabei keine Verschiebung der Koordinaten von X nach Y erfolgte. Um keine unnütze Verwirrung zu stiften, sind jedoch alle diese Programme nach einem einheitlichen Schema aufgebaut. Jetzt werden wir zum ersten Mal davon profitieren. Aber gehen wir lieber der Reihe nach vor.

7.5.1 Rund um das Dreieck

Es sind nicht sehr viele Formeln, die wir für die Berechnungen in den aufgeführten Beispielen benötigen. Alle haben sie jedoch eines gemeinsam: Sie drehen sich alle um das Dreieck. Speziell bei den Berechnungen in einem Koordinaten-System können wir uns auf das rechtwinklige Dreieck beschränken.

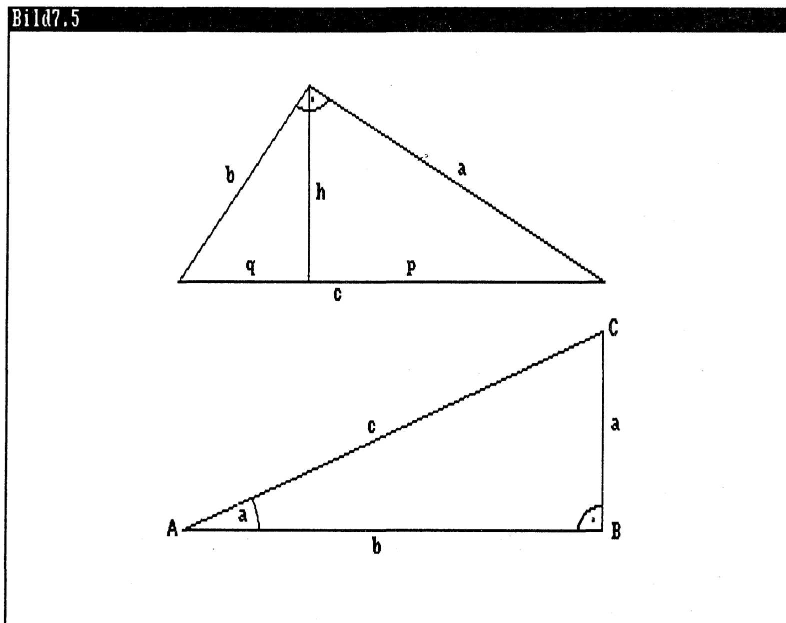


Bild 7.5: Das rechtwinklige Dreieck

Als rechtwinklige Dreiecke werden alle Dreiecke bezeichnet, bei welchen zwei Seiten im rechten Winkel, also in einem Winkel von 90 Grad, zueinanderstehen. Daß in jedem Dreieck alle Winkel zusammen 180 Grad ergeben, ist Ihnen sicher kein Geheimnis. Drei Sätze des rechtwinkligen Dreieckes sind wissenswert. Allgemeine Formeln:

Pythagoras	`c^2 = a^2 + b^2`
Höhensatz	`h^2 = p*q`
Kathetensatz	`a^2 = c*p`
	`b^2 = c*q`

Als nächstes nehmen wir uns die trigonometrischen Funktionen zur Brust. Auch dabei handelt es sich um Beziehungen der Seiten und der Winkel zueinander in einem rechtwinkligen Dreieck. Die einzelnen Buchstaben der Formeln finden Sie im Bild 7.5.

Winkelfunktionen:

```
sin(α)  = a/c   oder   a = sin(α) * c
cos(α)  = b/c   oder   b = cos(α) * c
tg(α)   = a/b
ctg(α)  = b/a
```

Am häufigsten werden davon die SIN und die COS-Funktion eingesetzt. Die Klammerschreibweise des Amiga-Basic wurde der Einfachheit halber übernommen. Besonders bei der Drehung haben wir es mit zwei verschiedenen Winkeln eines Grafikpunktes zu tun. Dazu benötigen wir die Beziehungen der Winkel untereinander.

Winkelfunktionen für Summe und Differenz zweier Winkel:

```
sin(a+b) = sin(a) * cos(b) + cos(a) * sin(b)
sin(a-b) = sin(a) * cos(b) - cos(a) * sin(b)
cos(a+b) = cos(a) * cos(b) - sin(a) * sin(b)
cos(a-b) = cos(a) * cos(b) + sin(a) * sin(b)
```

Mit diesen wenigen Formeln sind wir bestens gerüstet, um der Drehung auf den Leib zu rücken.

7.5.2 Drehung von Liniengrafiken

Zwei wichtige Faktoren beeinflussen die Drehung. Der erste ist der Drehwinkel, den ein Grafikpunkt zu seinem Ursprungswinkel im Koordinatenkreuz vollzieht. Fast ebenso wichtig ist der Abstand und die Lage des Drehpunktes, um den die Grafik rotiert. Um die Angelegenheit etwas übersichtlicher zu gestalten, wird in der folgenden Skizze nur die Veränderung an einem Punkt der Grafik gezeigt.

Das Koordinatenkreuz der Grafik ist so gelegt, daß sich der Rotationspunkt im Nullpunkt des Kreuzes befindet. Der ursprüngliche Punkt *P1* ist durch seine beiden Koordinaten *x* und *y* klar definiert. Den Drehwinkel *w* bestimmen wir selbst. Es bleiben also die beiden Unbekannten *x1* und *y1*. Zur Lösung des Problems greifen wir auf die Winkelfunktionen zurück.

```
a = sin(a) * c
b = cos(a) * c
```

Setzen wir in die Formeln die Buchstaben der Skizze ein, so sehen die beiden folgendermaßen aus:

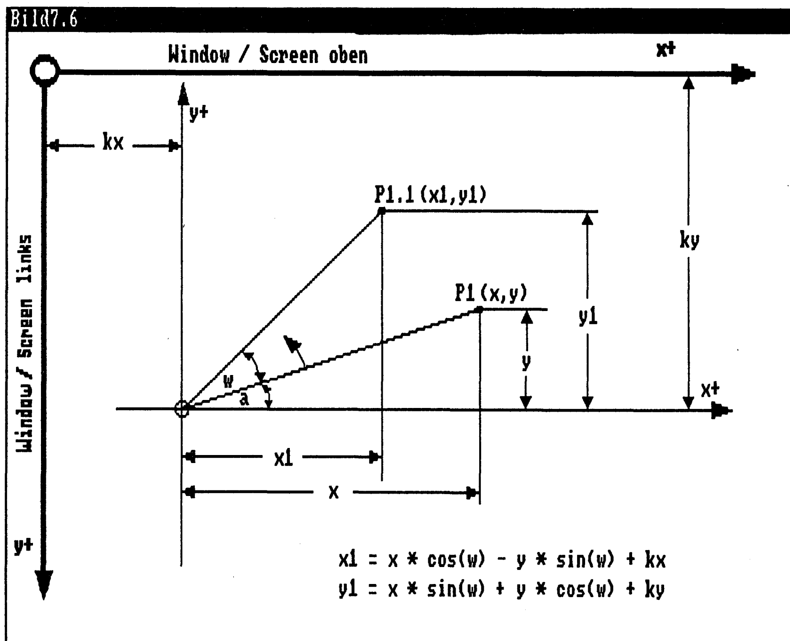


Bild 7.6: Drehung eines Grafikpunktes

```
x1 = cos (a+w)*r
y1 = sin (a+w)*r
```

Mit diesen beiden Formeln kommen wir wegen der jeweils 3 Unbekannten noch nicht weiter. Schauen wir einmal nach, ob wir noch weitere Formeln aus der kleinen Formelsammlung einsetzen können.

```
cos(a+b) = cos(a) * cos(b) - sin(a) * sin(b)
```

oder mit den Buchstaben der Skizze

```
cos(a+w) = cos(a) * cos(w) - sin(a) * sin(w)
sin(a+b) = sin(a) * cos(b) + cos(a) * sin(b)
```

oder mit den Buchstaben der Skizze

```
sin(a+w) = sin(a) * cos(w) + cos(a) * sin(w)
```

Setzen wir nun die beiden Formeln in die ersten beiden Formeln ein, sieht das schon ganz anders aus:

```
x1 = r*(cos(a) * cos(w) - sin(a) * sin(w))
y1 = r*(sin(a) * cos(w) + cos(a) * sin(w))
```

Um nun die restlichen Unbekannten eliminieren zu können, holen wir uns die Winkelfunktionen des Punktes *P1* zur Hilfe:

```
x = cos(a)*r  oder r = x/cos(a)
y = sin(a)*r  oder r = y/sin(a)
```

oder

```
y = sin(a)*x/cos(a)
x = cos(a)*y/sin(a)
```

Ersetzen wir nun *r* aus der letzten Formel mit den Funktionen aus der letzten Formel, so erhalten wir:

```
x1 = x/cos(a)*(cos(a) * cos(w) - sin(a) * sin(w))
```
oder
```
x1 = x * cos(w) - (x * sin(a) * sin(w)/cos(a))
```
oder
```
x1 = x * cos(w) - y * sin(w)
```

Die gleiche Prozedur mit der Koordinate *y1* schenken wir uns. Die Lösung lautet:

```
y1 = x * sin(w) + y * cos(w)
```

Bei der Umrechnung ist noch zu beachten, daß der Amiga einen Winkel im Bogenmaß fordert. Die Umrechnung ist nicht weiter aufregend:

```
Winkel im Bogenmaß = Winkel * Pi / 18Ø
```

Nachdem wir nun die größte Hürde übersprungen haben, überlegen wir uns noch, wie eine Verschiebung des Drehpunktes realisiert werden kann. Wie so oft, hilft auch hierbei eine kleine Skizze, das Verständnis zu erleichtern:

Im Bild 7.7 begnügen wir uns mit der Verschiebung einer einzigen Linie *L1*. Würden wir nur die Koordinaten-Variablen *kx* und *ky* verschieben, so würde die Grafik im gleichen Abstand um den neuen Mittelpunkt rotieren. Um die Rotationsachse zu verändern, müssen wir daher alle Koordinatenwerte der Grafik um die Verschiebungswerte *xw* und *yb* verändern (erhöhen oder reduzieren).

```
P1(x,y) -> P1.1(x+xw,y+yw)
```

Damit haben wir alle notwendigen Formeln erarbeitet, die für die Drehung von Liniengrafiken erforderlich sind. Die Berechnung ist von allen Transformationen die aufwendigste. Wie sieht nun das Ganze in einem Programm aus?

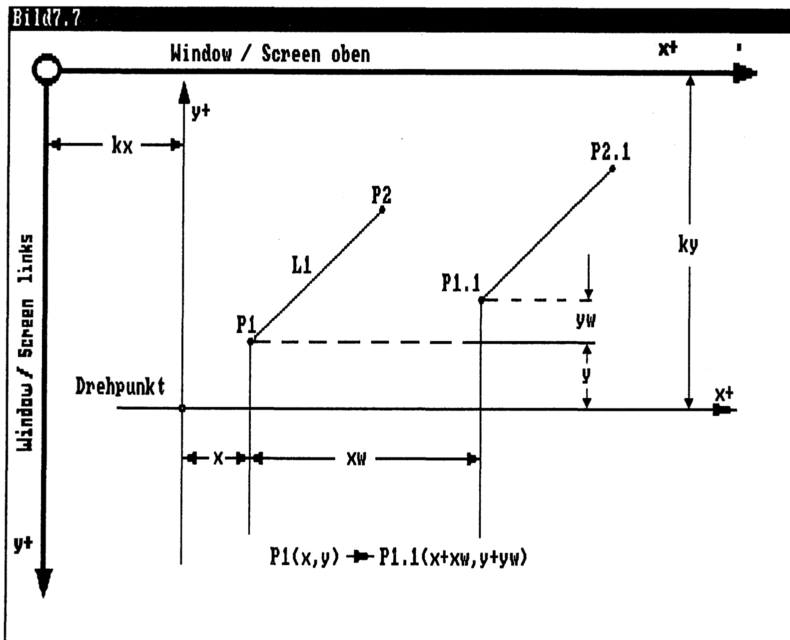


Bild 7.7: Verschieben des Drehpunktes

7.5.3 Das Programm rotieren

Wie bereits mehrfach vollzogen, zeichnen Sie zuerst die gewünschte Liniengrafik. Anschließend werden Sie aufgefordert, mit der Maus den Drehpunkt zu setzen. Dazu drücken Sie die linke Maustaste und halten sie fest. Es erscheint ein Fadenkreuz, das Sie beliebig über den Bildschirm verschieben können. An der gewünschten Position lassen Sie den Knopf los. Damit ist der Drehpunkt fixiert und wird durch ein kleines rotes Rechteck angezeigt. Um diesen Punkt beginnt die Grafik langsam zu rotieren. Die Rotationsgeschwindigkeit können Sie mit der Plus-Taste erhöhen und mit der Minus-Taste verringern. Zur Verschiebung des Drehpunktes betätigen Sie die Cursor-Tasten. Je nach der jeweiligen Pfeilrichtung der Cursor-Taste wird der Rotationspunkt um ein Pixel verschoben. Mit der Leertaste können Sie das Programm beenden.

```
REM rotieren  Pfad: Darstellung/7TransLinie/rotieren
'P7-4
'sx() Startpunkt Linie X-Achse
'ex() Endpunkt Linie X-Achse
'sy() Startpunkt Linie Y-Achse
'ey() Endpunkt Linie Y-Achse
'das Programm arbeitet mit 2 Screens und 1 Window
```

```
'
IF FRE(-1)<8ØØØØ& THEN BEEP:PRINT "Speicher zu klein":END
CLEAR
DEFINT a-z: d=4Ø
DIM sx(d),ex(d),sy(d),ey(d)
DIM co!(359),si!(359)
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"

Video:
  SCREEN 1,32Ø,2ØØ,2,1
  WINDOW 2,,(Ø,Ø)-(31Ø,18Ø),Ø,1
  scr1&=PEEKL(WINDOW(7)+46)
  WINDOW CLOSE 2
  SCREEN 2,32Ø,2ØØ,2,1
  WINDOW 3,,(Ø,Ø)-(31Ø,18Ø),Ø,2
  scr2&=PEEKL(WINDOW(7)+46)
  rp1&=scr1&+84
  rp2&=scr2&+84
  PALETTE 2,Ø,Ø,.8

winkel:
  pi!=3.141593:bo!=pi!/18Ø
  FOR w=Ø TO 359
    co!(w)=COS(w*bo!):si!(w)=SIN(w*bo!)
  NEXT

Netz
LOCATE 1,1:PRINT "bitte Objekt zeichnen"
LOCATE 2,1:PRINT "RETURN = fertig       "
i=-1
zeichnen:
  i=i+1:muss=MOUSE(Ø)
  WHILE MOUSE(Ø)=Ø
    ta-=INKEY-:IF ta-=CHR-(13)  THEN i=i-1:GOTO Mitte
  WEND
  muss=MOUSE(Ø):sx(i)=MOUSE(3):sy(i)=MOUSE(4)
  WHILE MOUSE(Ø)<>Ø :WEND
  ex(i)=MOUSE(5):ey(i)=MOUSE(6)
  LINE (sx(i),sy(i))-(ex(i),ey(i))
  IF i>=d THEN Mitte
  IF ta-<>CHR-(13) THEN zeichnen
```

```
Mitte:
  LOCATE 1,1:PRINT "Bitte den Mittelpunkt der "
  LOCATE 2,1:PRINT "Drehung mit der Maus markieren"
  muss=MOUSE(Ø):WHILE MOUSE(Ø)=Ø :WEND
  CALL SetDrMd&(WINDOW(8),3)
  WHILE MOUSE(Ø)<>Ø
    muss=MOUSE(Ø):x1=MOUSE(1):y1=MOUSE(2)
    LINE (x1,Ø)-(x1,WINDOW(3)),3
    LINE (x1,Ø)-(x1,WINDOW(3)),3
    LINE (Ø,y1)-(WINDOW(2),y1),3
    LINE (Ø,y1)-(WINDOW(2),y1),3
  WEND
  CALL SetDrMd&(WINDOW(8),1)

Koordinaten:
  fb=WINDOW(2):fh=WINDOW(3)
  kx=x1:ky=y1
  FOR n=Ø TO i
    sx(n)=sx(n)-kx:ex(n)=ex(n)-kx
    sy(n)=sy(n)-ky:ey(n)=ey(n)-ky
  NEXT

renderLeft=4:renderTop=2
skx=kx+renderLeft:sky=ky+renderTop
yanf=17:ymax=fh-yanf+renderTop:CLS
LOCATE 1,1:PRINT "Cursortasten = Drehpunkt verschieben"
LOCATE 2,1:PRINT "+ - = Tempo         Leertaste = ENDE"
xw=Ø:yw=Ø:te=1

Drehpunkt:
  FOR n=Ø TO i
    sx(n)=sx(n)+xw:ex(n)=ex(n)+xw
    sy(n)=sy(n)+yw:ey(n)=ey(n)+yw
  NEXT
  GOTO aktion

taste:ta-=INKEY-
  taste2: IF ta-<CHR-(27) OR ta->CHR-(45)THEN taste
  wahl= ASC(ta-)-27
  IF wahl=16 THEN te=te+1:GOTO aktion
  IF wahl=18 THEN te=te-1:GOTO aktion
  IF wahl <1 OR wahl>5 THEN taste
  xw=Ø:yw=Ø: ON wahl GOTO plusY,minusY,plusX,minusX,ende
```

```
plusY:  yw=yw+1: GOTO Drehpunkt
minusY: yw=yw-1: GOTO Drehpunkt
plusX:  xw=xw+1: GOTO Drehpunkt
minusX: xw=xw-1: GOTO Drehpunkt

aktion:
  a=-1:IF te>3Ø THEN te=3Ø
  IF te<1 THEN te=1
  WHILE a
    w=w+te:IF w>359 THEN w=Ø
    clearw (rp1&):CALL SetAPen&(rp1&,3)
    CALL RectFill&(rp1&,skx-1,sky-1,skx+1,sky+1)
    CALL SetAPen&(rp1&,1)
    FOR n=Ø TO i
      asx=sx(n)*co!(w)-sy(n)*si!(w)+skx
      IF asx<renderLeft THEN asx=renderLeft
      IF asx>WINDOW(2) THEN asx=WINDOW(2)
      asy=sx(n)*si!(w)+sy(n)*co!(w)+sky
      IF asy<yanf THEN asy=yanf
      IF asy>WINDOW(3) THEN asy=WINDOW(3)
      aex=ex(n)*co!(w)-ey(n)*si!(w)+skx
      IF aex<renderLeft THEN aex=renderLeft
      IF aex>WINDOW(2) THEN aex=WINDOW(2)
      aey=ex(n)*si!(w)+ey(n)*co!(w)+sky
      IF aey<yanf THEN aey=yanf
      IF aey>WINDOW(3) THEN aey=WINDOW(3)
      CALL Move&(rp1&,asx,asy)
      CALL Draw&(rp1&,aex,aey)
    NEXT
    CALL ClipBlit&(rp1&,renderLeft,yanf,rp2&,renderLeft,
**                                    yanf,fb,ymax,192)
    ta-=INKEY-:IF ta-<>"" THEN a=Ø
  WEND
GOTO taste2

ende:
  WINDOW CLOSE 3
  SCREEN CLOSE 2 :SCREEN CLOSE 1
  LIBRARY CLOSE
  ERASE sx,sy,ex,ey,co!,si!
END
```

```
SUB clearw (rp&) STATIC
  CALL SetRast&(rp&,Ø)
END SUB

SUB Netz STATIC
  FOR y = Ø TO WINDOW(3) STEP 5
    LINE (Ø,y)-(WINDOW(2),y),2
  NEXT
  FOR x = Ø TO WINDOW(2) STEP 5
    LINE (x,Ø)-(x,WINDOW(3)),2
  NEXT
END SUB
```

Zu Programmbeginn werden neben den jeweils 41 voreingestellten Koordinatenwerten noch zwei Felder mit jeweils 360 Feldelementen dimensioniert. Warum, werden Sie gleich erfahren. Nun folgen die Öffnungsmodalitäten für die Libraries. Zwei Bildschirme und ein Fenster werden geöffnet und die Adressen für die im Programm verwendeten Library-Routinen geholt. Das alles ist Ihnen inzwischen nicht mehr unbekannt.

Nun folgt auch schon die Erklärung für die beiden Felder mit den 360 Feldelementen. Die Zahl 360 steht für die 360 Grad eines Kreises. Bei der Sprungmarke »winkel« erfolgt für die SIN- und COS-Funktionen die Berechnung der Winkel von 0 bis 359 Grad. Die ermittelten Werte werden in den Feldvariablen *co!* und *si!* gespeichert. Damit erhöhen wir die Zeichengeschwindigkeit in der später folgenden Hauptschleife beträchtlich.

Nach der bekannten kleinen Zeichen-Routine werden beim Label »Mitte« die Werte für den Drehpunkt geholt. Dazu wird ein Koordinatenkreuz, das die Fensterabmessungen vollständig ausnützt, auf den Mauspfeil positioniert. Durch den Modus 3 der Routine *SetDrMd&* und durch ein zweimaliges Zeichnen der Koordinatenlinien kann das Kreuz beliebig auf dem Bildschirm verschoben werden. Hat der Anwender den Drehpunkt positioniert (MOUSE(0)=0), so wird der normale Modus wiederhergestellt.

Die Objekt-Koordinaten werden nun auf die Koordinaten des Drehpunktes eingestellt. Weiter geht es beim Label »Drehpunkt«, der auch bei jeder Veränderung des Drehpunktes angesprungen wird. Beim ersten Durchgang wird die Tastaturabfrage übersprungen und gleich zum Zeichnen des Objektes verzweigt.

Das Hauptprogramm läuft bei der Sprungmarke »aktion« ab. Zu Beginn wird die Tempo-Variable *te* auf die minimalen und maximalen Grenzen (1 bis 30 Grad) überprüft. In der WHILE/WEND-Schleife läuft das Zeichnen der Grafik in Schritten ab, die in der Variable *te* festgehalten sind. Die Temposteigerung bedeutet also nur, daß die Grafik während einer Umdrehung nicht so oft gezeichnet wird. Anschließend wird mit den Grafik-Routinen *SetAPen&* und *RectFill&* ein rotes Rechteck für den Drehpunkt gezeichnet.

In der FOR/NEXT-Schleife werden die einzelnen Linien der Grafik gezeichnet. Dazu werden die einzelnen Koordinaten der Anfangs- und Endpunkte berechnet, wie Sie es bereits im letzten Kapitel gesehen haben. Bei Unklarheiten blättern Sie am besten ein paar Seiten zurück. Damit die Original-Koordinaten erhalten bleiben, werden zum Zeichnen die Koordinaten der aktuellen Position in den Variablen *asx, asy, aex* und *aey* festgehalten. Nach jeder Berechnung der neuen Positionswerte folgt eine Prüfung auf eine eventuelle Überschreitung der Fenstergrenzen. Ist dies der Fall, so werden die Fenstergrenzen zum aktuellen Koordinatenwert. Am Ende der FOR/NEXT-Schleife werden die Linien mit *Move&* und *Draw&* in den unsichtbaren Screen gezeichnet. Damit kehrt das Programm in die WHILE/WEND-Schleife zurück. Mit der Super-Routine *ClipBlit&* wird die Grafik bzw. der Hintergrund-Screen in den sichtbaren Screen (bzw. Fenster) kopiert. Nun wird die Tastatur abgefragt.

Wurde in der Zwischenzeit eine Taste gedrückt, so wird die Schleife des Hauptprogrammes verlassen und zum Label »taste2« verzweigt. War die gedrückte Taste plus oder minus, so wird die Variable *te* für die Geschwindigkeit verändert und zur Hauptschleife zurückgesprungen. Die Cursortasten verändern die Variablen *xw* und *yw* für die Verschiebung des Drehpunktes um jeweils einen Pixel. Da die Variable ihren Wert behält, bedeutet ein mehrmaliges Drücken der gleichen Cursortaste eine Erhöhung des Verschiebeweges. Mit den neuen Werten wird, wie bereits erwähnt, zum Label »Drehpunkt« gesprungen. Dort werden sämtliche Koordinaten um die neuen Werte verändert.

Mit den neuen Koordinaten geht es zurück zu »aktion«. Nun wird das Objekt nach den neuen Werten gezeichnet, das heißt die Drehung erfolgt im neuen Rotationsabstand. Die Leertaste beendet mit den bekannten Schlußzeilen das Programm.

7.6 Schnelle Bilder, Bewegung mit PolyDraw

In der Einleitung des Kapitels über das Vergrößern und Verkleinern von Liniengrafiken versprach ich Ihnen, daß Sie damit auch einen Fahreffekt realisieren können. Das Versprechen will ich nun einlösen. Zum blitzschnellen Zeichnen der Grafik greifen wir auf die Library-Routine *PolyDraw* zurück, mit der wir schon die animierte Grafik *Kran* programmiert haben. Um die Geschwindigkeit weiter zu erhöhen, packen wir die Eckpunkte des Polygons in ein Variablen-Feld. Mit diesen Eigenschaften bestens gerüstet, demonstriert das Programm *UFO* durch ein gleichzeitiges Vergrößern und Verschieben einen sehr schnellen Fahreffekt. Es arbeitet so schnell, daß es in der Schleife des Hauptprogrammes sogar durch eine SOUND-Anweisung gebremst werden kann.

```
REM UFO  Pfad: Darstellung/7TransLinie/ufo
'P7-5
'Vergroessern und Verschieben
CLEAR
DEFINT a-z:DIM f(3):DIM bl(13,4Ø)
DIM we(255):FOR i=Ø TO 255:we(i)=RND*127:NEXT
WAVE Ø,we:ERASE we
LIBRARY ":bue/graphics.library"
SCREEN 1,64Ø,256,2,1: WINDOW 2,,,Ø,1
scr&=PEEKL(WINDOW(7)+46):rp&=scr&+84 :vp&=scr&+44
WINDOW CLOSE 2

farben: FOR i=Ø TO 3:READ f(i):NEXT
CALL LoadRGB4&(vp&,VARPTR(f(Ø)),4)
Objekt: FOR i=Ø TO 13:READ bl(i,Ø):NEXT

kx=16Ø              'X-Koordinate
fakt!=1             'Vergroesserung x und y
FOR n=1 TO 4Ø
  FOR i = Ø TO 13 STEP 2
    bl(i,n)=bl(i,Ø)*fakt! +kx
    ky=1Ø+n*2         'Y-Koordinate
    bl(i+1,n)=bl(i+1,Ø)*fakt!+ky
  NEXT i
  fakt!=fakt!+.8
NEXT n

CALL SetDrMd&(rp&,3)
FOR d = Ø TO 2Ø
  CALL Move&(rp&,bl(Ø,Ø),bl(1,Ø))
  CALL PolyDraw&(rp&,7,VARPTR(bl(Ø,Ø)))
  FOR n=1 TO 4Ø
    SOUND 1ØØ+8*n,.5,5Ø+5*n,Ø
    CALL Move&(rp&,bl(Ø,n),bl(1,n))
    CALL PolyDraw&(rp&,7,VARPTR(bl(Ø,n)))
    CALL Move&(rp&,bl(Ø,n-1),bl(1,n-1))
    CALL PolyDraw&(rp&,7,VARPTR(bl(Ø,n-1)))
  NEXT n
  CALL Move&(rp&,bl(Ø,n-1),bl(1,n-1))
  CALL PolyDraw&(rp&,7,VARPTR(bl(Ø,n-1)))
NEXT d
CALL SetDrMd&(rp&,1)
```

```
ende:
SCREEN CLOSE 1: ERASE f,bl
LIBRARY CLOSE :END

Farbwerte:
DATA &HE,&HFØØ,&hØfbØ,&hØf9Ø

Ufo:
DATA -5,Ø,-1,2,Ø,Ø,1,2,5,Ø,Ø,5,-5,Ø
```

Zuerst dimensionieren wir die Variablen und die Wellenform für ein weißes Rauschen. Nach der Öffnung der Grafik-Library wird ein neuer Bildschirm mit Fenster geöffnet, die benötigten Adressen ausgelesen und das Fenster wieder geschlossen. Das Programm zeichnet direkt in den neuen Screen. Die Farben werden eingelesen und eine neue Farbtafel mit *LoadRGB4* gesetzt. Die 14 Eckpunkte der Grafik selbst werden auch eingelesen. Das Polygon hat seinen Nullpunkt, also sein Koordinaten-Kreuz oben, in der Mitte der Grafik. Damit lassen sich die 40 Bildschirmpositionen aller Polygon-Koordinaten in den folgenden, ineinander geschachtelten FOR/NEXT-Schleifen leicht berechnen. Die äußere Schleife mit dem Schleifenzähler *n* steht für die Bildschirmposition der gesamten Grafik. Die innere Schleife mit dem Schleifenzähler *i* berechnet dann die einzelnen Eckpunkte der Grafik. Im Variablen-Feld *Bl()* folgen hintereinander eine X- und eine Y-Koordinate. Für die Vergrößerung werden die Anfangskoordinaten *Bl(i,0)* mit dem Vergrößerungsfaktor *fakt!* multipliziert. Zur X-Koordinate der Grafik wird die Entfernung zum linken Rand *kx* addiert und zur Y-Koordinate kommt die Entfernung zum oberen Screen-Rand hinzu.

Nachdem alle Bildschirmpositionen in dem Variablenfeld *Bl()* gespeichert sind, kann es losgehen. Das Programm ist auf 21 Schleifendurchgänge in der äußeren Schleife ausgelegt. In der inneren Schleife wird mit *Move* und *PolyDraw* die Grafik gezeichnet, die zuvor gezeichnete Grafik gelöscht, neu gezeichnet etc. Dazwischen sorgt die SOUND-Anweisung für die passende musikalische Untermalung. Nach der 40. Position kann der nächste Durchgang beginnen.

Mit diesem attraktiven Programm beenden wir die Transformation der Liniengrafik. Sicherlich haben Sie viele Anregungen für eigene Programme gewinnen können. Welche Möglichkeiten der Umformung von Grafiken die Pixelgrafiken bieten, zeigt Ihnen das anschließende Kapitel.

Kapitel 8

Umformung von Pixelgrafiken

Einige Male haben Sie inzwischen den Begriff Pixelgrafik gelesen. Was ist eigentlich damit gemeint? Nun, in dem Wort steckt nichts Geheimnisvolles. Es dient nur zur besseren Unterscheidung der Liniengrafiken von den normalen Standard-Grafiken. Da wir bei einer Veränderung der Standard-Grafik jedes einzelne Pixel umformen müssen, ist der Begriff sehr aussagefähig. Im Gegensatz dazu steht die Liniengrafik, wie im letzten Kapitel zu sehen war, wo nur die Linien, bzw. die Koordinaten ihrer Punkte, verändert werden mußten.

Auch für die Pixelgrafik gilt, daß für die Transformation der Grafiken das zu manipulierende Objekt von den Koordinaten des Bildschirmes zu lösen ist. Die Grafik erhält, wie bei der Liniengrafik, ein eigenes Koordinatensystem. Es erübrigt sich daher eine eigene Skizze. Sie können gegebenenfalls noch einen Blick auf das Bild 7.1 werfen. Der wesentliche Unterschied liegt darin, daß wir bei der Umformung der Pixelgrafik immer einen, meist rechteckigen, Bildschirmausschnitt verändern werden.

Fassen wir zusammen: Das Hauptproblem bei der Programmierung der Umwandlungen ist darin zu finden, daß von einem spezifizierten Bildschirmausschnitt jedes einzelne Pixel verändert werden muß. Normalerweise ist das eine zeitraubende Angelegenheit, die einem die Freude an der Manipulation der Grafiken vergällen kann. Das muß aber nicht sein! Sie werden in diesem Kapitel Methoden und Techniken kennenlernen, die genauso oder mindestens fast so schnell sind wie die Äquivalente der Liniengrafiken.

8.1 Verschieben von Flächen

Bereits die erste Transformation, das Verschieben von Pixelgrafiken, bestätigt die Aussage von soeben. Auch diese Art der Bildbeeinflussung, ist nicht komplizierter als das Beispiel der Liniengrafik, ist fast ein Kinderspiel. Wahrscheinlich haben Sie schon eigene Versuche in dieser Richtung unternommen, die dann sicherlich auch erfolgreich verlaufen sind. Die Erklärung der angewendeten Technik fällt daher kürzer als gewohnt aus.

8.1.1 Wie die Grafik verschoben wird

Wir legen wieder das Koordinatenkreuz an den zu verschiebenden Bildausschnitt an. Allerdings wählen wir als Nullpunkt die obere linke Ecke des Bildes. Der Grund liegt in der einfacheren Handhabung der verwendeten Anweisungen GET und PUT. Mit diesen starken Grafik-Befehlen des Amiga ist der Rest eine Kleinigkeit. Da sich am Prinzip der Verschiebung nichts ändert, können Sie wieder das Bild 7.2 der Liniengrafik zu Hilfe nehmen.

Der rechteckige Bereich, der verschoben werden soll, wird durch die Koordinatenpaare der linken oberen und der rechten unteren Ecke definiert. Mit der GET-Anweisung speichern wir die Bilddaten in einem Variablen-Feld:

```
GET (x1,y1)-(x2,y2),Bild&
```

Dazu muß natürlich vorher die Feld-Variable *Bild&* dimensioniert werden. Im Beispielprogramm wurde das Bild in einem Feld von langen Ganzzahlen (&), also als Langworte, gespeichert. Die Rechnung sieht so aus:

```
Inh=2+(y2-y1+1)*2*INT((x2-x1+16)/16)*tiefe/4
DIM Bild&(Inh)
```

Das Ergebnis wird, da ein Langwort 4 Byte enthält, durch 4 geteilt.

Das gespeicherte Bild kann nun mit PUT auf die Position gebracht werden. Die Ursprungsposition liegt im Abstand des Koordinatenkreuzes *kx* und *ky* der Grafik:

```
PUT (x1+kx,y1+ky),Bild&
```

Durch ein zweifaches Zeichnen auf die gleiche Position wird die Grafik wieder gelöscht. Die PUT-Anweisung hat den voreingestellten Modus XOR.

Wird die Grafik verschoben, müssen wir natürlich die Variablen für die Verschiebung *vx* und *vy* hinzufügen:

```
PUT (x1+kx+vx,y1+ky+vy),Bild&
```

Setzen wir diese wenigen Grundlagen gleich in einem hübschen Programm um.

8.1.2 Segeln mit Area Move

Das Programm zeichnet knapp zwei Dutzend Segelschiffe auf den Bildschirm. Mit der Maus können Sie nun markieren, mit welchem oder welchen Segelschiffen Sie über den Bildschirm segeln wollen. Solange Sie die Auswahltaste der Maus gedrückt halten, können Sie den durch ein flimmerndes Rechteck markierten Bereich noch ändern. Sobald aber die Taste losgelassen ist, wird das Bild gespeichert und der markierte Bildaus-

schnitt folgt jeder Mausbewegung. Sie können nun wählen, ob Sie einen neuen Start des Programmes mit einer anderen Grafik wünschen (RETURN-Taste), oder ob das Programm beendet werden soll.

```
REM AreaMove  Pfad: Darstellung/8TransPixel/AreaMove
'P8-1
'kx   Koordinatenkreuz des Objektes X-Achse
'ky   Koordinatenkreuz des Objektes Y-Achse
'vx   Verschiebung X-Achse
'vy   Verschiebung Y-Achse
'
DEFINT a-z
LIBRARY ":bue/graphics.library"
SCREEN 1,320,256,3,1
WINDOW 2,"AreaMove",,0,1
win2&=WINDOW(7)
PALETTE 7,.6,.6,.6
READ m
DIM x(m),y(m)
FOR j=0 TO m
  READ x(j),y(j)
NEXT j

start:
CLS
FOR schleife = 0 TO 20
  RANDOMIZE TIMER
  fakt!=RND :IF fakt!<.2 THEN fakt! = .2
  px=RND*220:py=(RND*200)+30
  fa=7*RND:COLOR fa
  FOR pic = 0 TO m
    AREA (px+x(pic)*fakt!,py+y(pic)*fakt!)
  NEXT pic
  AREAFILL
NEXT schleife
COLOR 3
LOCATE 1,1:PRINT "bitte den zu verschiebenden Bereich "
LOCATE 2,1:PRINT "mit der Maus markieren"
muss=MOUSE(0):WHILE MOUSE(0)=0 :WEND
x1=MOUSE(3):y1=MOUSE(4)
CALL SetDrMd&(WINDOW(8),3)
WHILE MOUSE(0)<>0
```

```
    muss=MOUSE(Ø):x2=MOUSE(1):y2=MOUSE(2)
    LINE (x1,y1)-(x2,y2),3,b:LINE (x1,y1)-(x2,y2),3,b
WEND
CALL SetDrMd&(WINDOW(8),1)
IF x1>x2 THEN SWAP x1,x2
IF y1>y2 THEN SWAP y1,y2
Dimensionierung:
Inh=2+(y2-y1+1)*2*INT((x2-x1+16)/16)*3/4
DIM Bild&(Inh)
GET (x1,y1)-(x2,y2),Bild&

Koordinaten:
kx=x1:ky=y1      'Nullpunkt oben links
x1=Ø :y1=Ø       'Objekt im Koordinatenkreuz

verschieben:
LOCATE 1,1:PRINT "Das Objekt folgt jeder Mausbewegung"
LOCATE 2,1:PRINT "RETURN = Neustart    Taste = fertig"
px1=PEEKW(win2&+14)
py1=PEEKW(win2&+12)

aktion:
px2=PEEKW(win2&+14)
py2=PEEKW(win2&+12)
vx=px2-px1 :vy=py2-py1      'Verschiebungsvektoren
ta-=INKEY-: IF ta-<>"" THEN ende
IF vx=Ø AND vy=Ø THEN aktion   'keine Mausbewegung
PUT (x1+kx,y1+ky),Bild&
PUT (x1+kx+vx,y1+ky+vy),Bild&
kx=kx+vx:ky=ky+vy
px1=px2:py1=py2
IF ta-="" THEN aktion

ende:
IF ta-=CHR-(13) THEN ERASE Bild& :GOTO start
WINDOW CLOSE 2: SCREEN CLOSE 1
LIBRARY CLOSE
ERASE Bild&,x,y
END

Schiff:
DATA 19
DATA Ø,Ø,2Ø,1Ø,8Ø,1Ø,9Ø,5,6Ø,5,6Ø,Ø,52,Ø
```

```
DATA 52,-5,6Ø,-5,6Ø,-25,52,-25,52,-3Ø,48,-3Ø,48,-25
DATA 35,-25,35,-5,48,-5,48,Ø,4Ø,Ø
DATA Ø,Ø
```

Nach dem Öffnen der Grafik-Library wird ein neuer Bildschirm für 8 Farben mit einem Fenster erstellt. Anschließend geht es gleich mit dem Zeichnen eines Grafik-Bildes weiter. (An dieser Stelle können Sie auch eine eigene Grafik einlesen oder zeichnen.) Für die Grafik verwendet das Programm ein stilisiertes Segelschiff, dessen Daten nun eingelesen und in den Feldvariablen *x()* und *y()* gespeichert werden. Ab dem Label »start« werden 21 Segelschiffe gezeichnet. Damit die Grafik etwas pep bekommt, werden die Segelschiffe in den 8 zur Verfügung stehenden Farben, die durch Zufall gesetzt werden, auf den Bildschirm gebracht. Daß dabei auch Schiffe in der Hintergrundfarbe gezeichnet werden, wird in Kauf genommen. Auch die Größe der Schiffe wird durch eine Zufallszahl *fakt!* bestimmt. Mit diesem Faktor *fakt!*, dessen Wert zwischen 0,2 und 1 liegt, werden die Daten-Werte innerhalb der AREA-Anweisung verändert. Natürlich wird auch die Position der Schiffe auf dem Bildschirm durch Zufallszahlen ermittelt. Da mit RANDOMIZE TIMER der Zufallsgenerator bei jedem Schleifendurchlauf neu gestellt wird, können Sie also bei jedem Aufruf des Programmes eine andere Grafik erwarten.

Nach dem Zeichnen der Grafik wartet das Programm, nach einer entsprechenden Anweisung an den Anwender, auf eine Betätigung der linken Maustaste. Wurde die Taste gedrückt, so wird die Position des Mauszeigers, zu diesem Zeitpunkt, aus MOUSE(3) und MOUSE(4) geholt und in den Variablen *x1* und *x2* gemerkt. Die folgende Schleife (WHILE MOUSE(0)<>0) wird so lange durchlaufen, wie die Maustaste niedergehalten bleibt. Der Zeichenmodus ist währenddessen, mit der Library-Routine *SetDrMd&*, auf JAM1+COMPLEMENT gesetzt. Durch das zweimalige Zeichnen des Rechteckes in der Schleife wird es gezeichnet und anschließend wieder gelöscht. Die Koordinaten der unteren rechten Ecke des Rechteckes werden mit MOUSE(1) und MOUSE(2) abgefragt und an *x2* und *y2* übergeben.

Wenn die Maustaste nicht mehr gedrückt ist, wird die Schleife verlassen und mit *SetDrMd* der normale Zeichenmodus wiederhergestellt. Die Variablen des markierten Bereiches werden überprüft und, wenn notwendig, vertauscht. Bevor das ausgewählte Rechteck mit GET gespeichert werden kann, muß die Dimensionierung vorgenommen werden. Bei »Dimensionierung« wird der benötigte Speicherbereich berechnet und an die Variable *Inh* übergeben. Nach der Dimensionierung in *Bild&* kann endlich die Grafik mit GET zwischengespeichert werden.

Bei »Koordinaten« werden die Koordinaten der Grafik festgelegt. Bei einem rechteckigen Bildausschnitt ist das kein Problem. Wie bereits praktiziert, werden bei der Sprungmarke »verschieben« die X- und Y-Werte des Mauszeigers aus der Window-Struktur gelesen. Jetzt geht es wieder rund. Die Hauptschleife des Programmes beginnt beim

Label »aktion«. Für die aktuellen Koordinaten des Mauszeigers muß wieder die Window-Struktur herhalten. Die Differenz der alten (*px1* und *py1*) und der neuen Koordinaten *px2* und *py2* gibt die Verschiebung des Koordinatenkreuzes des Objektes wieder. Die Variablen *vx* und *vy* enthalten die Verschiebungswerte. Ist ihr Inhalt Null, so hat keine Verschiebung stattgefunden und das Programm verzeigt zu »aktion« zurück. Andernfalls wird mit PUT und den ursprünglichen Koordinatenwerten das alte Bild gelöscht. Der voreingestellte Wert bei PUT ist XOR, das heißt, ein zweimaliges Zeichnen löscht das Objekt, ohne den Hintergrund zu zerstören. (Wie der Zeichenmodus 3, den wir zum Zeichnen des Rechteckes eingesetzt hatten.) Mit einer weiteren PUT-Anweisung, bei der die Koordinaten um die Verschiebungswerte *vx* und *vy* erhöht werden, wird die Grafik auf die neue Position gebracht. Nun müssen nur noch die Ausgangsvariablen ihre neuen Werte erhalten, und der nächste Schleifendurchlauf kann beginnen.

Das Verschieben der Grafik kann durch einen Tastendruck beendet oder neu gestartet werden. Vor einem neuen Start wird mit ERASE der Bildinhalt freigegeben, damit es nicht zu einer doppelten Dimensionierung kommt. Sicherlich fallen Ihnen für Ihre Programme weitere effektvolle Anwendungsmöglichkeiten ein, bei der Sie die Verschiebung von Grafiken einsetzen können.

8.2 Spiegeln

Mit der Spiegelung von Pixelgrafiken können Sie in vielen Programmen besonders verblüffende Effekte einbauen. Mit relativ geringem Aufwand kann zum Beispiel der Titel Ihres Programmes in besonders schönen Buchstaben und darunter in Spiegelschrift erscheinen. Oder sähe es nicht toll aus, wenn sich in der Abenddämmerung die Grafik in einem See spiegelt? Natürlich läßt sich auch die Spiegelung bei der Konstruktion komplexer Grafiken besonders zeitsparend einsetzen. Auch Ihnen werden auf Anhieb eine ganze Menge von Anwendungsmöglichkeiten einfallen. Bevor wir aber weiter von den tollen Möglichkeiten träumen, sollten wir erst anschauen, wie es gemacht wird.

8.2.1 Flächenspiegelung an zwei Achsen

Bei der Spiegelung eines Grafikbildes können wir das bei der Liniengrafik Gelernte kaum einsetzen. Bei einer Pixelgrafik haben wir nun mal keine Linien, die wir manipulieren können. Die Änderung der Vorzeichen beim Vorgang des Spiegelns gilt natürlich auch für die Pixelgrafik. Es scheint, als gäbe es nur einen Ausweg, und zwar, daß jeder einzelne Grafikpunkt ausgelesen und auf die neu berechnete Position gesetzt wird. Das dauert uns aber viel zu lange. Schließlich muß unser Grafikspezialist Amiga eine Möglichkeit kennen, damit auch bei einem Grafikbild die Spiegelung sofort sichtbar wird.

Wenn wir uns überlegen, daß ein Grafikbild aus lauter Zeilen besteht und auch eine zeilenweise Spiegelung zum richtigen Ergebnis führen müßte, kommen wir der Sache schon näher. Zur besseren Darstellung hilft meistens eine kleine Skizze:

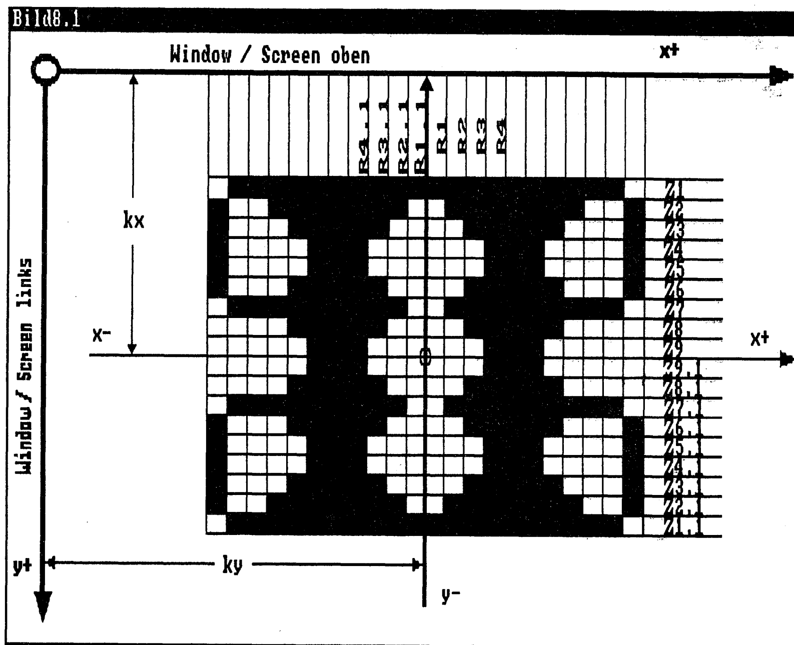


Bild 8.1: Spiegelung einer Pixelgrafik

Aus dem Bild 8.1 können Sie erkennen, wie die zeilenweise Spiegelung vor sich gehen könnte. Für die Spiegelung an der X-Achse müßte die Zeile Z1 an die Position der Reihe Z1.1 gebracht werden, die Grafikzeile Z2 an die Position der Reihe Z2.1 etc. Die Grafik von Bild 8.1 ist 9 Zeilen hoch. Damit wären 9 solcher Operationen notwendig. Nicht viel anders sieht es bei der Spiegelung an der Y-Achse aus. Man müßte dann, anstatt das Bild zeilenweise zu verschieben, jede einzelne Spalte umsetzen. Die Spalte R1 würde an die Position der Spalte R1.1 gebracht, die Spalte R2 käme zur Position der Spalte R2.1 etc.

Soweit hört sich ja alles ganz gut an. Doch wie bekommen wir die einzelnen Reihen oder Spalten an die richtige Position? Nun, wir unterteilen die Grafik einfach in lauter kleine Teilgrafiken. Bei der Spiegelung an der X-Achse sind das bei unserem Beispiel 9 Teilgrafiken, die alle nur eine Grafikzeile hoch sind, aber die volle Breite behalten. Bei der Spiegelung an der Y-Achse gehen wir analog vor. Die Teilgrafiken sind dann 1 Pixel breit und behalten die volle Höhe.

Damit können wir für jede Teilgrafik die normalen Grafikbefehle des Amiga einsetzen. Mit GET lesen wir den Inhalt einer Zeile (oder Spalte) ein und bringen ihn mit PUT an die neue Position. Die X- und Y-Werte verändern sich dabei, wie wir es bei der Liniengrafik erfahren haben. Wenn wir dabei die ausreichende Dimensionierung beachten, funktioniert die Spiegelung damit einwandfrei.

Das folgende Programm lief zu Testzwecken auch mit dieser Variante. Die Spiegelung dauerte dabei, je nach der Größe des Grafikbildes, etwa eine bis maximal drei Sekunden. Im Verhältnis zum Lesen und Setzen einzelner Bildpunkte ist das eine wahnsinnige Beschleunigung. Kann es noch schneller gehen? Sie werden staunen. Es geht wirklich noch schneller. Dabei hilft uns wieder einmal eine Routine der Library.

Sie haben sicherlich schon erraten, welche Routine ich meine. Natürlich spreche ich von der superschnellen Kopier-Routine *ClipBlit*. Auch beim Einsatz der erwähnten Library-Routine können wir genauso vorgehen, wie wir es vorher besprochen haben. Die Grafik wird also in Teilgrafiken zerlegt. Nur an die Stelle der beiden Anweisungen GET und PUT tritt eine einzige Routine: *ClipBlit*. Betrachten wir uns dazu im Detail die Spiegelung an der X-Achse.

Das zu spiegelnde Rechteck ist durch die Variablen *sx1* und *sy1* für die obere linke Ecke und durch die Veränderlichen *sx2* und *sy2* definiert. Die Höhe der Grafik ist in *h* gespeichert. Nun kopieren wir die erste Grafikzeile des Ursprungsbildes (*sx1,sy2*) in die letzte Zeile der zu erstellenden Grafik (*sx1,sy2+h*). Es folgt die zweite Zeile (*sx1,sy2+1*), die in die vorletzte Zeile (*sx1,sy2+h-1*) des zukünftigen Spiegelbildes kopiert wird, etc. Die Schleife, die die komplette Grafik spiegelt, könnte dann folgendes Aussehen haben:

```
i=sy2+h
FOR z=sy1 TO sy2
  i=i-1
  CALL ClipBlit&(rp2&,sx1,z,rp2&,sx1,i,b,1,192)
NEXT
```

Die Routine

```
ClipBlit&(rp2&,sx1,z,rp2&,sx1,i,b,1,192)
```

erhält dabei folgende Werte zugewiesen:

```
rp2&= Adresse des RastPorts, Quelle
sx1 = linke (obere) Ecke der Teilgrafik X-Koordinate, Quelle
z   = linke (obere) Ecke der Teilgrafik Y-Koordinate, Quelle
rp2&= Adresse des RastPorts, Ziel
sx1 = linke (untere) Ecke der Teilgrafik X-Koordinate, Ziel
i   = linke (untere) Ecke der Teilgrafik Y-Koordinate, Ziel
b   = Breite der Teilgrafik
1   = Höhe der Grafik, eine Zeile
192 = Minterm, es wird alles kopiert
```

Die Bezeichnungen linke untere oder linke obere Ecke haben bei einer Grafik, die nur eine Zeile hoch ist, natürlich nur einen theoretischen Sinn. Wie schnell die Routine ist, davon können Sie sich bei dem folgenden Programm selbst überzeugen.

8.2.2 Way via XY

Als Grafik für die Spiegelung werden beim Start des Programmes drei Flugzeuge auf den Bildschirm gebracht. Das Programm bittet nun, den zu spiegelnden Bereich mit der Maus zu markieren. Am besten geeignet dafür ist das rechte obere Viertel des Fensters. Sobald der Bereich markiert ist, erscheint ein Requester und fragt, ob das Objekt an der X-Achse gespiegelt werden soll. Wird *ja* angeklickt, so verschwindet der Requester und die Spiegelung wird durchgeführt. Der nächste Requester wird durch einen Druck auf die Auswahltaste der Maus herbeigerufen. Nun können noch die Spiegelungen an der Y-Achse und an beiden Achsen angewählt werden. Der letzte Requester fragt, ob ein weiterer Versuch gewünscht wird. Wenn ja, so wird das Programm neu gestartet, ansonsten beendet.

```
REM WAY via XY  Pfad: Darstellung/8TransPixel/WAYviaXY
'11.8.88
'P8-2
CLEAR
DEFINT a-z

LibraryOeffnen:
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION AutoRequest&() LIBRARY
  LIBRARY ":bue/graphics.library"
  LIBRARY ":bue/intuition.library"

SpeicherReservieren:
  art&=3+(2^16)
  rek&=0:rk&=VARPTR(rek&)
  m1&=AllocRemember&(rk&,100,art&)
  IF m1&=0 THEN PRINT "memory-FEHLER":BEEP:GOTO ende

Video:
  SCREEN 1,320,256,3,1
  WINDOW 2,"WAY VIA XY   YX AIV YAW",,16,1
  win2&=WINDOW(7)
  rp2&=WINDOW(8)
  scr&=PEEKL(WINDOW(7)+46)
  PALETTE 7,.6,.6,.6
```

```
start:
  txt3-="ja":txt4-="nein"
  CLS
  FOR d=Ø TO 2
    spx=1ØØ*d:spy=7Ø+(5*d)
    READ m
    FOR j=Ø TO m
     READ n,fa,x,y :COLOR fa
      AREA (spx+x,spy+y)
      FOR i=Ø TO n
        READ x,y
        AREA STEP (x,y)
      NEXT i
      AREAFILL
    NEXT j
    RESTORE Flugzeug
  NEXT d

Auftakt:
  LOCATE 1,1:PRINT "bitte den zu spiegelnden Bereich "
  LOCATE 2,1:PRINT "mit der Maus markieren"
  muss=MOUSE(Ø):WHILE MOUSE(Ø)=Ø :WEND
  x1=MOUSE(3):y1=MOUSE(4)
  WHILE MOUSE(Ø)<>Ø
    muss=MOUSE(Ø):x2=MOUSE(1):y2=MOUSE(2)
    CALL SetDrMd&(WINDOW(8),3)
    LINE (x1,y1)-(x2,y2),3,b:LINE (x1,y1)-(x2,y2),3,b
  WEND
  CALL SetDrMd&(WINDOW(8),1)
  LINE (x1-1,y1-1)-(x2+1,y2+1),2,b
  IF x1>x2 THEN SWAP x1,x2
  IF y1>y2 THEN SWAP y1,y2
  sx1=x1
  sx2=x2
  sy1=y1
  sy2=y2
  h=sy2-sy1
  b=sx2-sx1
  LOCATE 1,1:PRINT SPACE-(33)
  LOCATE 2,1:PRINT SPACE-(22)
```

```
spiegeln:
  txt1-="Soll das Objekt um die "
  txt2-="X-Achse gespiegelt werden?"
  GOSUB Requester
  IF jn& THEN GOSUB spiegelnX
  txt1-="Soll das Objekt um die "
  txt2-="Y-Achse gespiegelt werden?"
  GOSUB Requester
  IF jn& THEN GOSUB spiegelnY
  txt1-="Soll das Objekt um beide "
  txt2-="Achsen gespiegelt werden?"
  GOSUB Requester
  IF jn& THEN GOSUB spiegelnXY
  txt1-="Wollen Sie einen weiteren"
  txt2-="Versuch wagen?"
  GOSUB Requester
  IF jn& THEN start
ende:
  WINDOW CLOSE 2
  SCREEN CLOSE 1
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  LIBRARY CLOSE
END
spiegelnX:
  i=sy2+h+2
  'sichern der Fenstergrenze Y
  IF i>WINDOW(3) THEN BEEP:RETURN
  FOR z=sy1 TO sy2
    i=i-1
    CALL ClipBlit&(rp2&,sx1,z,rp2&,sx1,i,b,1,192)
  NEXT
  GOTO Koordinaten
spiegelnY:
  i=sx1+1
  'sichern der Fenstergrenze X
  IF i-b <Ø THEN BEEP:RETURN
  FOR z=sx1 TO sx2
    i=i-1
    CALL ClipBlit&(rp2&,z,sy1,rp2&,i,sy1,1,h,192)
  NEXT
  GOTO Koordinaten
```

```
spiegelnXY:
  i=sy2+h+1
  'sichern der Fenstergrenze Y
  IF i>WINDOW(3) THEN BEEP:RETURN
  FOR z=sy1 TO sy2
    i=i-1
    CALL ClipBlit&(rp2&,sx1,z,rp2&,sx1,i,b,1,192)
  NEXT
  i=sx1+1:di=sy1+h
  'sichern der Fenstergrenze X
  IF i-b <Ø THEN BEEP:RETURN
  FOR z=sx1 TO sx2
    i=i-1
    CALL ClipBlit&(rp2&,z,di,rp2&,i,di,1,h,192)
  NEXT

Koordinaten:
  LINE (Ø,y2)-(WINDOW(2),y2),3
  LINE (x1,Ø)-(x1,WINDOW(3)),3
  warten:
  muss=MOUSE(Ø):WHILE MOUSE(Ø)=Ø :WEND
RETURN

Requester:
  CALL Texte (txt1-,Ø,Ø,m1&+2Ø)
  CALL Texte (txt2-,1Ø,2Ø,n&)
  CALL Texte (txt3-,Ø,4Ø,n&)
  CALL Texte (txt4-,Ø,6Ø,n&)
  jn&= AutoRequest&(WINDOW(7),m1&,m1&+4Ø,m1&+6Ø,Ø,Ø,25Ø,6Ø)
RETURN

SUB Texte (text-,y%,o%,n&) STATIC
  SHARED m1&
  text-=text-+CHR-(Ø):POKE m1&+o%,1
  POKE m1&+o%+2,2     :POKEW m1&+o%+4,5 :POKEW m1&+o%+6,5+y%
  POKEL m1&+o%+12,SADD(text-)           :POKEL m1&+o%+16,n&
END SUB

Flugzeug:
DATA 3
DATA 2,4
DATA Ø,Ø,5Ø,Ø,Ø,25,-5Ø,-25
DATA 2,5
```

```
DATA 6Ø,Ø,5Ø,Ø,-5Ø,25,Ø,-25
DATA 5,7
DATA 4Ø,-4Ø,3Ø,Ø,Ø,5,-1Ø,1Ø,-1Ø,Ø,-1Ø,-1Ø,Ø,-5
DATA 7,6
DATA 55,5Ø,-3,-5,-2,-12,Ø,-5Ø,5,-2Ø
DATA 5,2Ø,Ø,5Ø,-2,12,-3,5
```

Das Programm entspricht in seinem Aufbau dem Programm »LineMirror« aus dem Kapitel über die Liniengrafiken. Wir werden daher nur die Punkte betrachten, die geändert wurden.

Der Screen erhält die Tiefe 3 für 8 Farben. Aus den Window-Funktionen 7 und 8 werden die Adressen für die Fenster- und die RastPort-Struktur geholt. Das aktuelle Programm arbeitet nur mit einer Bitmap, da ein Double Buffering nicht erforderlich ist.

Beim Label »start« werden in drei mal vier Schleifendurchgängen die Daten für drei Bilder ausgelesen, mit AREA gezeichnet und mit AREAFILL ausgemalt. Die Bilder stellen drei Flugzeuge dar, die zur besseren Erkennung der Spiegelung aus jeweils vier Farben bestehen. Selbstverständlich können Sie an dieser Stelle jede X-beliebige Grafik einlesen, zum Beispiel mit dem Programm »IFFuniversal«, oder auch zeichnen.

Nun wartet das Programm, bis die linke Maustaste (MOUSE(0)<>0) gedrückt wird. Beginnend von diesem Startpunkt (x1=MOUSE(3) und y1=MOUSE(4)) wird ein Rechteck zur aktuellen Position des Mauspfeiles gezogen. Da der Zeichen-Modus 3 gesetzt ist und das Rechteck zweimal mit diesem Modus gezeichnet wird, kann es ohne Zerstörung des Hintergrundes vergrößert oder verkleinert werden. Sobald die Maus-Taste losgelassen ist, wird das Rechteck mit dem Modus 1 gezeichnet. Dadurch kann der Anwender den Vorgang der Spiegelung besser verfolgen.

Anschließend werden die Variablen *x1, x2, y1* und *y2* geprüft. Wurden bei der Auswahl des zu spiegelnden Bereiches die Anfangs- und Endpunkte des Rechteckes vertauscht, so würden bei der Berechnung der Höhe und der Breite negative Werte entstehen. Bei der eingesetzten Library-Routine bedeutet das einen Absturz des Rechners. Mit SWAP verhindern wir diese Unannehmlichkeit. Die Variablen *sx1* bis *sy2* erhalten die Screen- bzw. Bitmap-Werte von *x1* bis *y2*.

Die Label »spiegeln« und »ende« kennen Sie vom Vorprogramm aus dem Kapitel über die Liniengrafik. Nun kommen wir zum Kern des Programmes. Die Spiegelung um die X-Achse beginnt bei »spiegelnX«. Die Variable *i* erhält die Grafikzeile, auf die die erste Zeile des Bildes gespiegelt wird. Da die Routine nur Werte innerhalb der Bitmap zuläßt, müssen die Screen-Grenzen vor dem Überschreiben geschützt werden. Ist die Variable *i* größer als WINDOW(3), wird daher die Spiegelung nicht ausgeführt und mit einem Warnsignal zurückgesprungen. In der Schleife wird von der Oberkante *sy1* bis zur Unterkante *sy2* des ausgewählten Bereiches die Spiegelung, wie bereits erklärt, aus-

geführt. Am Ende der Subroutine wird zur Sprungmarke »Koordinaten« verzweigt. Dort werden, wie bei der Bezeichnung nicht anders zu erwarten ist, die Spiegel-Koordinaten gezeichnet.

Bei »spiegelnY« vollzieht sich der soeben geschilderte Vorgang in der Y-Richtung. Etwas komplizierter wird es bei der Spiegelung um beide Achsen in der Subroutine »spiegelnXY«. Eine direkte Spiegelung kann mit der Library-Routine nicht durchgeführt werden. Es wird daher zuerst an der X-Achse gespiegelt und von dort eine zweite Spiegelung durchgeführt. Soll dieser Zwischenschritt für den User unsichtbar bleiben, bietet es sich an, dafür eine eigene Bitmap anzulegen. Die Technik werden wir bereits im nächsten Programm besprechen.

Der Rest des Programmes zeigt nichts Neues. Sie sehen, daß auch die Spiegelung kompletter Grafiken ohne Schwierigkeiten und dazu sehr schnell programmiert werden kann. Es gibt also keinen Hinderungsgrund, diese Technik nicht in Ihren eigenen Programmen einzusetzen.

8.3 Flächen unter die Lupe genommen

Das Verkleinern und Vergrößern von Linien konnten wir mit relativ einfachen Mitteln in einem schnellen Programm verwirklichen. Daß es bei der Pixelgrafik meistens etwas komplizierter wird als bei der Liniengrafik, haben Sie inzwischen selbst erfahren. Natürlich wollen wir trotzdem eine möglichst schnelle Ausführung der Transformation realisieren.

Auch die Lupenfunktion bietet viele interessante Anwendungsmöglichkeiten. Das kann man übrigens von allen Transformationen dieses Kapitels behaupten. Schließlich können Sie damit jede beliebige Grafik bearbeiten, ganz egal ob es sich dabei um Ihr eigenes Konterfei oder um das Bild der Mona Lisa handelt.

8.3.1 Der Weg der Lupe durch drei Bitmaps

Die Grundregel Nummer 1 für Mehrfachtransformationen muß natürlich auch hier beachtet werden. Sämtliche Manipulationen müssen, um Verzerrungen und Verfälschungen zu vermeiden, von der ursprünglichen Grafik ausgehen. Da wir nicht, wie bei der Liniengrafik, die Bilddaten in einigen Variablen-Feldern speichern können, müssen wir dafür einen eigenen Speicherbereich zur Verfügung stellen. Aber damit nicht genug. Um auf einigermaßen vernünftige Geschwindigkeiten zu kommen, werden wir wieder unsere blitzschnelle Routine *ClipBlit* einsetzen. Mit dieser Routine können wir, wie Ihnen sicherlich noch von der Spiegelung her in Erinnerung ist, nur in zwei Schritten die Grafik verändern. Das bedeutet, daß wir zum Vergrößern oder Verkleinern in der X-Achse einen weiteren Speicherbereich benötigen. Wir nehmen dazu im Beispiel-

programm einen kompletten Screen, obwohl eine, dem Bild angepaßte Bitmap weniger Speicherplatz beanspruchen würde. Um ein Double Buffering zu realisieren, müßten wir sogar noch einen weiteren Speicherblock zur Transformation der Y-Achse einrichten. Von dort könnte dann das Bild in den letzten bzw. sichtbaren Screen kopiert werden. Wir werden auf das Double Buffering verzichten. Die Veränderung der Y-Achse wird damit dem Anwender sichtbar. Im Klartext bedeutet das, daß das transformierte Bild zeilenweise die alte Grafik überschreibt.

Bevor wir auf die Einzelheiten eingehen, werfen Sie bitte einen Blick auf die folgende Skizze, in der die geschilderte Reihenfolge der Bilderstellung zu sehen ist:

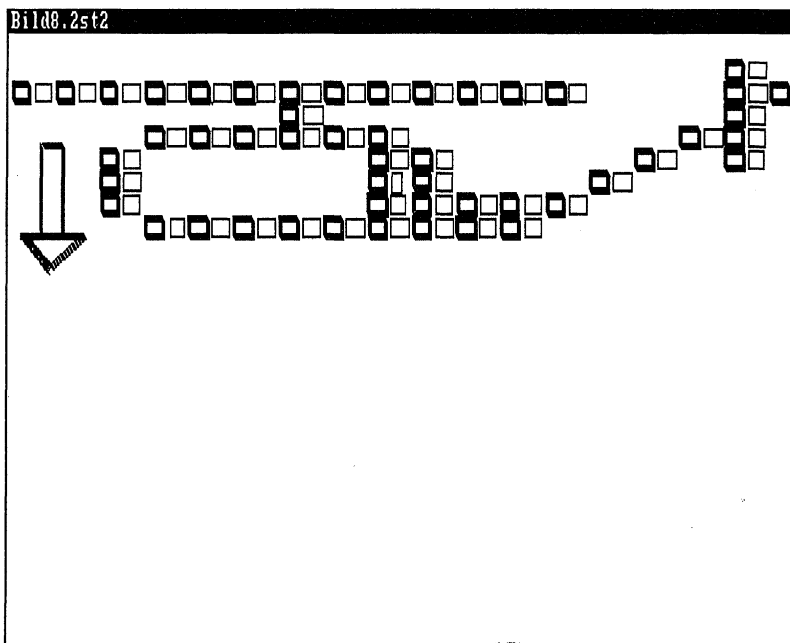


Bild 8.2: Lupenfunktion einer Pixelgrafik

Stufe 1
Eine Custom-Bitmap speichert das Originalbild.

Stufe 2
Im Screen 1 wird das Bild in der X-Achse verändert.

Stufe 3
Im sichtbaren Screen 2 wird die Grafik in der Y-Richtung geändert und fertiggestellt.

Die Reihenfolge der Transformationen im Bild 8.2 stellt eine Verdoppelung der Größe in X- und Y-Richtung dar. Die starke Vergrößerung wurde gewählt, damit das System, das dahintersteckt, besser erkennbar wird. Sie sehen daran, daß durch die Vergrößerung in jeder Grafikspalte eine leere Spalte und in jeder Grafikzeile eine Leerzeile entstanden ist. Da eine Vergrößerung natürlich nicht mehr zeigen kann, als das Original an Details enthält, werden die Leerräume mit den danebenliegenden Pixeln gefüllt. Bei einer Verkleinerung dagegen werden Grafikzeilen und -spalten herausgenommen. Das bedeutet, um bei dem gewählten Schema zu bleiben, bei einer Verkleinerung um die Hälfte wird jede zweite Grafikspalte und -zeile entfernt. Bei einer Verkleinerung wird daher die Qualität des Bildes erheblich schlechter.

Diese grobe Zusammenfassung bedarf einer detaillierteren Betrachtung. Zur Ausführung der Transformation wenden wir wieder die Technik der Teilgrafiken an. Die Grafik wird also in einzelne Spalten (X-Achse) oder einzelne Zeilen (Y-Achse) zerlegt. Als Beispiel betrachten wir uns dazu die Vergrößerung um die X-Achse. Nehmen wir einmal an, die Grafik sei 21 Pixel breit, 4 Bildpunkte hoch und der Vergrößerungsfaktor (vx!) sei 1,3. Die erste Spalte bleibt auf ihrer Position: x=0, neue Position: x*vx!, oder 0*1,3=0. Die zweite Spalte behält ebenfalls ihre Position: x=1, neue Position: 1*1,3=1. Bei der dritten Spalte erhalten wir die erste Veränderung: x=2, neue Position: 2*1,3=3. Im Zusammenhang, einer vier Bildpunkte hohen Grafik, sieht das Ganze folgendermaßen aus:

```
neue Position    Ø  2 4  6 8
x=x*1,3           1  3 5  7  9

Grafik-Pixel     XX XXXX XXX XXX XXX XXXX XX
                 XX XXXX XXX XXX XXX XXXX XX
                 XX XXXX XXX XXX XXX XXXX XX
                 XX XXXX XXX XXX XXX XXXX XX

alte Position    Ø 2 4 6 8
x=Position        1 3 5 7 9
```

Die Anzahl und die Größe der leeren Spalten richtet sich nach dem Wert des Vergrößerungsfaktors. Diese Leerspalten werden mit den Farbwerten der letzten gezeichneten Spalte gefüllt. Bei der Vergrößerung um die Y-Achse wird analog dem gezeigten Schema vorgegangen. Am besten setzen wir das neue Wissen gleich in einem praktischen Beispiel ein.

8.3.2 Das Programm »Magnify«

Mit dem folgenden Programmbeispiel können Sie die Grafik wiederum bis auf die Größe eines Pixels verkleinern. Die Cursortasten für die Vergrößerung zaubern dann wieder das vollständige Bild herbei.

```
REM Magnify  Pfad: Darstellung/8TransPixel/Magnify
'P8-3
'vx   Vergroesserungsfaktor X-Achse
'vy   Vergroesserungsfaktor Y-Achse
'das Programm arbeitet mit 2 Screens, 1 BitMap und 1 Window
'
IF FRE(-1)<110ØØØ& THEN PRINT "Speicher zu klein":BEEP:END
CLEAR
DEFINT a-z
GOSUB LibraryOeffnen
GOSUB SpeicherReservieren :IF fehl THEN ende

Video:
  tiefe=3
  SCREEN 1,32Ø,2ØØ,tiefe,1
  WINDOW 2,,(Ø,Ø)-(31Ø,18Ø),Ø,1
  scr1&=PEEKL(WINDOW(7)+46):WINDOW CLOSE 2
  SCREEN 2,32Ø,2ØØ,tiefe,1
  WINDOW 3,,(Ø,Ø)-(31Ø,18Ø),Ø,2
  scr2&=PEEKL(WINDOW(7)+46)
  rp1&=scr1&+84
  rp2&=scr2&+84
  mb2&=PEEKL(rp2&+4)
  vp2&=scr2&+44

farben:
  FOR i=Ø TO 7:READ f(i):NEXT
  CALL LoadRGB4&(vp2&,VARPTR(f(Ø)),8)

Prolog:
  GOSUB Bildeinlesen
  h=73 :ob=72 :b=65 :li=122
  kx=1Ø :ky=2Ø
  fb=WINDOW(2):fh=WINDOW(3)
  renderLeft=4:renderTop=2              'Fensterrand
  skx=kx+renderLeft:sky=ky+renderTop    'Screenabmessung
beruecksichtigen
  yanf=17:ymax=fh-yanf+renderTop
  GOSUB Bitmapanlegen :IF fehl THEN ende
  'RastPort fuer BitMap3 anlegen
  CALL InitRastPort&(rp3&)
  POKEL rp3&+4,mb&
  CALL ClipBlit&(rp2&,li,ob,rp3&,Ø,Ø,b,h,192)
```

```
vergroessern:
  CLS:vx!=1:vy!=1
  LOCATE 1,1:PRINT "Cursortasten zum Vergroessern/Verkleinern"
  LOCATE 2,1:PRINT "Leertaste = ENDE"
  taste:ta-=INKEY-:IF ta-<CHR-(28) OR ta->CHR-(32) THEN taste
  wahl= ASC(ta-)-27
  ON wahl GOTO vergrY,verklY,vergrX,verklX,ende

vergrY:vy!= .9*vy! :GOTO size
verklY:vy!=1.1*vy! :GOTO size
vergrX:vx!=1.1*vx! :GOTO size
verklX:vx!= .9*vx!

size:
  clearw rp1&:altxp=skx
  FOR n=Ø TO b-1
    neux=n*vx!:xp=skx+neux
    IF xp>=WINDOW(2) THEN BEEP:n=b:GOTO taste
    CALL ClipBlit&(rp3&,n,Ø,rp1&,xp,sky,1,h,192)
    IF xp-altxp >Ø THEN
      FOR ff=altxp TO xp-1
        CALL ClipBlit&(rp3&,n,Ø,rp1&,ff,sky,1,h,192)
      NEXT
    END IF
    altxp=xp+1
  NEXT
  altyp=sky
  FOR n=Ø TO h-1
    neuy=n*vy!:yp=sky+neuy :pbm=n+sky
    IF yp>=WINDOW(3) THEN BEEP:n=h:GOTO taste
    CALL ClipBlit&(rp1&,5,pbm,rp2&,5,yp,31Ø,1,192)
    IF yp-altyp >Ø THEN
      FOR ff=altyp TO yp-1
        CALL ClipBlit&(rp1&,5,pbm,rp2&,5,ff,31Ø,1,192)
      NEXT
    END IF
    altyp=yp+1
  NEXT
  LINE (Ø,yp-1)-(WINDOW(2),WINDOW(3)),Ø,bf
  GOTO taste
```

```
ende:
  FOR i = Ø TO tiefe-1
    IF bp&(i) THEN CALL FreeRaster&(bp&(i),b,bmRows)
  NEXT
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  WINDOW CLOSE 3 :SCREEN CLOSE 2 :SCREEN CLOSE 1
  IF fehl THEN
    ON fehl GOSUB Fe1,Fe2
    BEEP:PRINT ft-
  END IF
  LIBRARY CLOSE
END

Fe1:ft-= "Fehler beim Einlesen des Bildes":RETURN
Fe2:ft-= "Speicher nicht ausreichend":RETURN

LibraryOeffnen:
  DECLARE FUNCTION AllocRaster&() LIBRARY
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION xOpen&() LIBRARY
  DECLARE FUNCTION xRead&() LIBRARY
  LIBRARY ":bue/graphics.library"
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/dos.library"
RETURN

SpeicherReservieren:
  art&=3+(2^16)
  rek&=Ø:rk&=VARPTR(rek&)
  m1&=AllocRemember&(rk&,1ØØ,art&)
  IF m1&=Ø THEN fehl=2:RETURN
  m2&=AllocRemember&(rk&,8ØØ,art&)
  IF m2&=Ø THEN fehl=2:RETURN
  adr&=m2&
  mb&=AllocRemember&(rk&,1ØØ,art&) 'Fuer BitMap
  IF mb&=Ø THEN fehl=2:RETURN
  rp3&=AllocRemember&(rk&,8Ø,art&) 'fuer RastPort3
  IF rp3&=Ø THEN fehl=2
RETURN

Bildeinlesen:
  fina-=":Bild/MrBrown"+CHR-(Ø)
  offen&=xOpen&(SADD(fina-),1ØØ5)
```

```
  IF offen&=Ø THEN fehl=1:RETURN
  lese&=xRead&(offen&,adr&,12)
  suchen=-1
  WHILE suchen
    lese&=xRead&(offen&,adr&,8)
    chlg&=PEEKL(adr&+4)
    IF chlg&/2-FIX(chlg&/2)>Ø THEN chlg&=chlg&+1
    text-="":FOR i=Ø TO 3
    text-=text-+CHR-(PEEK(adr&+i)):NEXT
    IF text-="BODY" THEN suchen=Ø:GOTO wtr
    chunk%=chlg&
    rest&=xRead&(offen&,adr&,chunk%)
    wtr:
  WEND
  bbreite=79 :bhoehe=84 :bytprozeile=bbreite/8
  bitmap1&=mb2&+8 :diff=12Ø/8+(7Ø*32Ø/8)
  FOR h= Ø TO bhoehe-1
    FOR eb=Ø TO tiefe-1
      zeile&=PEEKL(bitmap1&+(4*eb))+4Ø*h+diff
      lese&=xRead&(offen&,zeile&,bytprozeile)
  NEXT eb,h
  IF offen& THEN CALL xClose&(offen&)
RETURN

Bitmapanlegen:
  bmBytPerRow=b/8 'Bytes per Grafikzeile
  bmRows=h        'Grafik-Zeilen
  bmDepth=3       'Tiefe
  volum&=bmBytPerRow*bmRows
  FOR i = Ø TO bmDepth-1
    bp&(i)=AllocRaster&(b,bmRows)
    IF bp&(i)=Ø THEN fehl=1:RETURN
    CALL BltClear&(bp&(i),volum&,Ø)
  NEXT
  POKEW mb&,bmBytPerRow         'Bytes/Reihe
  POKEW mb&+2,bmRows            'Reihen
  POKE mb&+5,bmDepth            'Tiefe
  FOR n = Ø TO bmDepth-1
    POKEL mb&+8+(n*4),bp&(n)    'n BitPlanes
  NEXT
  CALL InitBitMap&(mb&,bmDepth,b,bmRows)
RETURN
```

```
SUB clearw (rp&) STATIC
  CALL SetRast&(rp&,Ø)
END SUB

Farbwerte:
DATA &HE,&HFØØ,&hØfbØ,&hØf9Ø
DATA &hØe9Ø,&hØc9Ø,&hbØ,&hØØØ
```

Wie üblich definieren wir zu Programmbeginn die Library-Funktionen und öffnen die Bibliotheken. Die folgenden Speicherreservierungen benötigen wir zum Einlesen mit Hilfe der DOS-Library für einen neuen RastPort und für eine Bitmap-Struktur. Anschließend öffnen wir die beiden Screens mit einer Farbtiefe 3 für 8 Farben. Die Übergabe der verschiedenen Speicheradressen an die einzelnen Variablen kennen Sie vom Vorprogramm. Zusätzlich holen wir die ViewPort-Adresse aus der Screen-Struktur des sichtbaren Bildschirmes. Diese Adresse benötigen wir bereits für die nachfolgende Library-Routine *LoadRGB4&*. Mit ihr holen wir uns die DATAs für die Farbtabelle des ViewPorts.

Beim Label »Bildeinlesen« lesen wir das Grafikbild *MrBrown* in den sichtbaren Screen ein. Das Bild ist im IFF-Standard gespeichert. Diesen Programmabschnitt können Sie jedoch kaum für Ihre eigenen Programme verwenden, da damit nur die reinen Bilddaten eingelesen werden. Die anderen Parameter des IFF-Bildes werden vom Programm gesetzt. Ich habe diesen Weg gewählt, um das Programm möglichst kurz zu halten. Weitere Informationen und ein komplettes Lade- und Speicherprogramm für Grafiken finden Sie in dem entsprechenden Kapitel.

Bei der anschließenden Variablenfestlegung werden wieder die unterschiedlichen Abmessungen von Screen und Window berücksichtigt. Dann wird zur Subroutine »Bitmapanlegen« gesprungen. Sie stellt eine Bitmap in der Größe der Grafik zur Verfügung. Für diese neue Bitmap legen wir mit *InitRastPort* einen eigenen RastPort an. Damit kann nun gefahrlos das Bild, das sich noch im Screen 2 befindet, in die neu angelegte Bitmap kopiert werden. Den Programmabschnitt beim Label »vergroessern« haben wir bereits beim entsprechenden Programm der Liniengrafik besprochen.

Ab »size« wird's wieder spannend. Den unsichtbaren Bildschirm löschen wir in der gewohnten Manier. Zuerst kommt die Veränderung der X-Achse an die Reihe. Ausgangspunkt ist die ursprüngliche Grafik. Die Schleife zählt daher die einzelnen Spalten der Grafik aus der Anwender-Bitmap. Die Variable *neux* erhält dabei die neue X-Position der Grafik-Spalte zugewiesen. Die tatsächliche Position auf dem Bildschirm wird in *xp* festgehalten. Sobald die tatsächliche Position den erlaubten Bereich des Fensters verläßt, wird das Zeichnen der Grafik abgebrochen (xp> =WINDOW(2)). Mit der Routine *ClipBlit* kann jetzt das ursprüngliche Bild aus der Anwender-Bitmap spaltenweise an die richtige Position des Screens Nummer 1 gezeichnet werden. Bei jedem Durchlauf merkt sich das Programm die vorhergehende X-Position in *altxp*. Dadurch kann durch

```
IF xp-altxp >Ø
```

ermittelt werden, ob eine Leerspalte entstanden ist. In diesen Fällen werden in einer zweiten FOR/NEXT-Schleife die leeren Spalten aufgefüllt.

Nicht anders ist die Vorgehensweise bei der Veränderung der Y-Werte. Beim Kopieren von Screen 1 auf den vorderen Bildschirm wird jedoch nicht die Breite des Bildes aus Screen 1 kopiert, sondern die komplette Fensterbreite. Damit wird bei einer Verkleinerung um die X-Achse der Rest des zuletzt gezeichneten Bildes gleich mit gelöscht und bleibt nicht als unliebsames Fragment auf dem Bildschirm zurück. Auch bei einer Verkleinerung um die Y-Achse muß der Rest des vorhergehenden Bildes vernichtet werden. Dazu zeichnen wir einfach ein Rechteck in der Hintergrundfarbe ab der letzten Y-Position bis zu den Fenstergrenzen.

Das war's schon. Die Renovierungsarbeiten am Schluß fallen wegen der Speicherfreigaben und wegen einer kleinen Fehler-Routine etwas größer aus, als bisher gewohnt. – Sie werden mir zustimmen, daß alles in allem gesehen auch die Vergrößerung oder Verkleinerung einer Pixelgrafik so kompliziert gar nicht ist. Als Lupenfunktion sollte sie in keinem anspruchsvollen Grafikprogramm fehlen.

8.4 Flächenrotation

Bisher ist es uns gelungen, die Transformationen in teilweise atemberaubender Geschwindigkeit durchzuführen. Sicherlich sind Sie der Meinung, daß es bei der Drehung ganzer Flächen wohl oder übel damit vorbei sein wird, ja vorbei sein muß. Natürlich liegen Sie mit Ihrer Vermutung richtig. Aber obwohl Sie recht haben, werden Sie, und soviel will ich Ihnen schon verraten, im Beispielprogramm unseren Mr. Brown mit einer solchen Geschwindigkeit um sich selbst rotieren sehen, daß Sie ihn nicht erkennen können.

8.4.1 Jedes Pixel an seinen Platz

Das Prinzip der Drehung kennen Sie bereits von der Drehung der Liniengrafiken her. Dort hatten wir jedoch nur für jede Linie zwei Punkte zu drehen. Trotzdem traten dabei bereits Probleme der ordentlichen Darstellung, hervorgerufen durch das unterschiedliche Pixelverhältnis x zu y und durch die Auflösung des Bildschirmes, auf. Dieses Problem wird uns bei der Drehung ganzer Flächen natürlich erst recht zu schaffen machen.

Betrachten wir uns dazu einen Programmausschnitt, der eine Grafik um einen bestimmten Winkel nach rechts gedreht zeichnet. Wenn Sie sich nicht mehr an das Prinzip erinnern können, schlagen Sie bitte im Kapitel über die Drehung von Linien-Grafiken nach.

```
FOR y=Ø TO 71
  FOR x=Ø TO 71
    col=ReadPixel&(rp1&,x,y)
    CALL SetAPen&(rp1&,col)
    asx=x*COS(Winkel)-y*SIN(Winkel)+kx
    asy=x*SIN(Winkel)+y*COS(Winkel)+ky
    erg&=WritePixel&(rp1&,asx,asy)
NEXT x,y
```

Um diese Drehung zu zeichnen, benötigt das Programm ca. 60 Sekunden. Dabei ist zu berücksichtigen, daß die Winkel selbst bereits vorher berechnet wurden. Der Zeitaufwand ist nicht verwunderlich. Immerhin müssen bei der aufgeführten Grafik 72 * 72 = 5184 Grafikpunkte farblich bestimmt, die neuen Koordinaten x und y berechnet und schließlich auf die neue Position gezeichnet werden. Leider ist damit die neu gezeichnete Grafik immer noch nicht zu gebrauchen. Hervorgerufen durch das unterschiedliche Verhältnis der Pixel in X- und Y-Richtung bleiben auf der Zeichnung eine Reihe nicht gesetzter Punkte bestehen. Ein häßlicher Anblick.

Was kann man dagegen tun? Eine Möglichkeit ist die, jede Grafikreihe etwas versetzt ein zusätzliches Mal zu zeichnen. Der abgeänderte Programmausschnitt hätte dann folgendes Aussehen:

```
FOR y=Ø TO 71
  FOR x=Ø TO 71
    col=ReadPixel&(rp1&,x,y)
    CALL SetAPen&(rp1&,col)
    asx=x*COS(Winkel)-y*SIN(Winkel)+kx
    asy=x*SIN(Winkel)+y*COS(Winkel)+ky
    erg&=WritePixel&(rp1&,asx,asy)
    asx=(x+.5)*COS(Winkel)-(y+.5)*SIN(Winkel)+kx
    asy=(x+.5)*SIN(Winkel)+(y+.5)*COS(Winkel)+ky
    erg&=WritePixel&(rp1&,asx,asy)
NEXT x,y
```

Die grafische Darstellung der Zeichnung nach obigem Programmausschnitt ist nicht gerade berauschend, aber das Bild ist deutlich und ohne gravierende Fehler gezeichnet. Der Preis dafür ist hoch. Fast 2 Minuten dauert nun das Zeichnen der Grafik.

Wenn in einem Programm nur eine Drehung einer Grafik benötigt wird, kann oder muß man mit dieser Zeit leben. Wollen Sie aber eine oder mehrere ganze Drehungen programmieren, so wird der Anwender wenig Freude daran haben. Rechnen wir als Minimum für eine Drehung mit 8 Grafikbildern, so benötigt das Programm 7*2=14 Minuten. Entsetzlich!

Um diese wahnsinnig langen Zeiten zu reduzieren, müssen wir kombiniert fast alle Techniken der Transformation einsetzen, die wir bisher besprochen haben. Durch Verschieben, Spiegeln, Kopieren und Umformen lassen sich die Ausführungszeiten beträchtlich abkürzen. Um zu sehen, wie die Kombination der einzelnen Techniken praktisch eingesetzt werden kann, betrachten wir uns nun die Drehung einer Grafik, bestehend aus 8 Einzelbildern. Sie entstammt dem Programm *turn* aus dem nächsten Kapitel. Dieses Programm benötigt zum Zeichnen der 7 bzw. 8 Grafiken nur 2 Minuten und 40 Sekunden. Eine ganz schöne Beschleunigung, wenn man bedenkt, daß zum Zeichnen des 45-Grad-Bildes davon schon fast 2 Minuten benötigt werden. Ist die Zeichenphase beendet, so geht es natürlich noch schneller. Die einzelnen Grafiken können dann mit einer Geschwindigkeit von bis zu 46 Grafiken pro Sekunde abgerufen werden! Wenn das kein Tempo ist...

Bevor diese Geschwindigkeit erreicht werden kann, muß natürlich ein klein wenig Arbeit investiert werden. Die 8 Einzelbilder, die für die Drehung benötigt werden, zeichnen wir in einen eigenen, für den Anwender nicht sichtbaren Screen. Bei Bedarf können wir sie der Reihe nach auf den sichtbaren Screen abrufen (Double Buffering). Wie die einzelnen Grafiken in dem unsichtbaren Schirm abgelegt werden, entnehmen Sie bitte der folgenden Skizze:

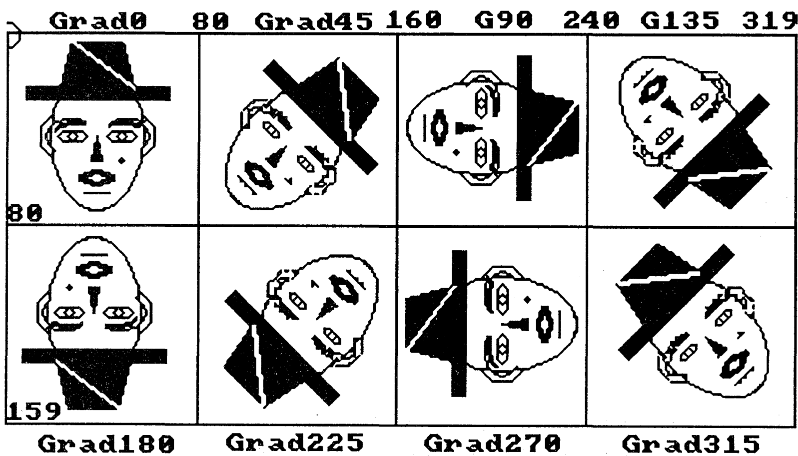


Bild 8.3: Speicherung von Einzelgrafiken für eine Drehung

Sie sehen, daß bei diesem Beispiel der Bildschirm in 8 gleich große Felder von 80 Pixel Breite und 80 Bildpunkten Höhe aufgeteilt ist. In jedes dieser Felder wird eine andere Ansicht der ursprünglichen Grafik gespeichert. Wichtig für die spätere Ausgabe ist dabei, daß die Bilder exakt so gespeichert werden, wie sie auf dem Bildschirm erscheinen sollen. Eine Korrektur der Lage muß also bereits bei der Ablage auf dem Hintergrund-Screen erfolgen.

Bild 1 (Grad0)
Hier hinein wird das ursprüngliche Bild gespeichert. Dazu holen wir, mit dem blitzschnellen Kopierbefehl *ClipBlit*, den Mr. Brown vom sichtbaren Bildschirm. *Grad0* dient gleichzeitig als Vorlage für die einzelnen Transformationen.

Bild 2 (Grad45)
Wie bereits ausführlich besprochen, wird das 2. Bild um 45 Grad gedreht gezeichnet. Alle Transformationen des Programmes *turn* drehen das Bild um seinen Mittelpunkt. Bei der eingesetzten Formel wird jedoch eine Drehung um die linke untere Ecke des Bild-Rechteckes durchgeführt. Um das Bild auf die richtige Position zu bekommen, müssen wir es entsprechend korrigieren. Die erste Korrektur betrifft die X-Achse. Bei der Drehung um 45 Grad eines Rechteckes (Bild) stellt die neue Breite des Bildes die Diagonale des Rechteckes dar. Bei einer Grafik von 72*72 Bildpunkten sind das 102 Pixel (SQR(72^2+72^2)). Da der Mr.Brown die Ecken des Rechteckes nicht ausfüllt, genügen trotzdem die 80 Pixel des Speicherbereiches. Wir zentrieren daher einfach das gedrehte Bild in die Mitte des ursprünglichen Bildes

```
asx=x*COS(Winkel)-y*SIN(Winkel)+8Ø+h/2
```

Auch die Y-Position muß korrigiert werden. Dazu schieben wir am Schluß die gedrehte Grafik um den entsprechenden Korrekturwert nach oben.

Bild 3 (Grad90)
Die Drehung um 90 Grad geht ebenfalls pixelweise vor sich. Der erste Punkt vom ersten Bild der ersten Grafik-Reihe wird zum ersten Punkt der letzten Grafik-Spalte, der zweite Punkt der Reihe kommt an die zweite Position der rechten Spalte etc. Es folgt die zweite Reihe, die zur zweiten Spalte von rechts wird. Da dabei keine Berechnung erforderlich ist, geht das Zeichnen der Grafik in zwei ineinandergeschachtelten Schleifen recht flott von sich.

Bild 4 (Grad135)
Um das Bild 2 um 90 Grad zu drehen, können wir leider nicht mit der Spiegelung arbeiten, da die Grafik sonst seitenverkehrt gezeichnet würde. Wir wenden daher die gleiche Technik an, wie wir sie bereits bei Bild 3 angewendet haben.

Bild 5 (Grad180)
Ab diesem Bild können wir mit schnellen Techniken die Grafiken im Sekundenbruchteil zeichnen. Bild 5 wird in zwei Schritten gezeichnet. Zuerst spiegeln wir Bild 1 um die X-Achse in das Feld von Bild 8, das als Puffer dient. Vom Puffer 8 spiegeln wir anschließend die seitenverkehrte Grafik an der Y-Achse in den Speicherbereich von Bild 5.

Bild 6 (Grad225)
Um Bild 6 zu zeichnen, wenden wir das gleiche Verfahren wie bei Bild 5 an. Dabei gehen wir von Bild 2 aus. Als Puffer dient wieder der Speicherbereich von Bild 8.

Bild 7 (Grad270)
Als Ausgangsgrafik wird Bild 3 gespiegelt. Das Übrige haben wir bereits bei Bild 5 und 6 besprochen.

Bild 8 (Grad315)
Auch das letzte Bild wird durch eine Doppelspiegelung gezeichnet. Bild 4 dient dieses Mal als Basis der Spiegelung. Leider haben wir keinen Puffer mehr frei. Wir degradieren daher einfach das Bild 1 zum Puffer und führen dort unsere Spiegelung durch. Am Schluß kopieren wir einfach wieder das Bild vom sichtbaren Screen in den Speicherbereich von Bild 1.

Sie haben an den einzelnen Beschreibungen gesehen, daß tatsächlich nur eine Grafik (45 Grad) auf die konventionelle, langsame Art und Weise gezeichnet werden mußte. Alle anderen Bilder konnten, meistens sehr schnell, durch Spiegelung oder andere Techniken gezeichnet werden. Für schnelle Drehungen reichen, wie Sie gleich sehen werden, 8 Einzelbilder völlig aus, um eine gleichförmige Drehung darzustellen. Bei langsameren Drehungen läßt sich das menschliche Auge natürlich nicht mehr überlisten. 16 oder gar 32 Einzelbilder sind dann für eine saubere Darstellung erforderlich. Je mehr Bilder Sie einsetzen, um so größer ist der Speicherbedarf. Damit der Anwender nicht zu lange auf das Zeichnen der einzelnen Grafiken warten muß, können Sie natürlich auch den Satz gezeichneter Grafiken abspeichern und bei Bedarf einlesen.

8.4.2 Turn, programmierte Drehung

Sicherlich wollen Sie die graue Theorie endlich in einem praktischen Beispiel umgesetzt sehen. Ich will Sie nicht länger auf die Folter spannen. Das Programm *turn* zeigt Ihnen all das, was ich vorher versprochen habe.

```
REM turn  Pfad: Darstellung/8TransPixel/turn
'P8-4
'das Programm arbeitet mit 2 Screens und 1 Window
'
CLEAR
IF FRE(-1)<5ØØØØ& THEN PRINT "Speicher zu klein":END
DEFINT a-z
GOSUB LibraryOeffnen
GOSUB SpeicherReservieren :IF fehl THEN Fehler2
GOSUB SchirmFenster
```

```
GOSUB Farben
GOSUB Bildeinlesen :IF fehl THEN Fehler1
GOSUB BilderZeichnen
CLS

Rotation:
a=-1
WHILE a
  Bild
  Taste
WEND

ende:
IF rk& THEN CALL FreeRemember&(rk&,-1)
WINDOW CLOSE 3 :SCREEN CLOSE 2 :SCREEN CLOSE 1
IF fehl THEN
  BEEP:BEEP:BEEP:PRINT ft$
  t&=TIMER:WHILE t&+5>TIMER:WEND
END IF
LIBRARY CLOSE :END

Fehler1:ft$= "Fehler beim Einlesen des Bildes":fehl=-1:GOTO ende
Fehler2:ft$= "Speicher nicht ausreichend":fehl=-1:GOTO ende

LibraryOeffnen:
DECLARE FUNCTION ReadPixel&() LIBRARY
DECLARE FUNCTION WritePixel&() LIBRARY
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION xOpen&() LIBRARY
DECLARE FUNCTION xRead&() LIBRARY
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"
LIBRARY ":bue/dos.library"
RETURN

SpeicherReservieren:
art&=3+(2^16):rek&=0:rk&=VARPTR(rek&)
m2&=AllocRemember&(rk&,800,art&)
IF m2&=0 THEN fehl=-1
adr&=m2&
RETURN
```

```
SchirmFenster:
tiefe=3:sw=320:sh=200: SCREEN 1,sw,sh-40,tiefe,1
WINDOW 2,,,0,1 :scr1&=PEEKL(WINDOW(7)+46)
rp1&=scr1&+84 :mb1&=PEEKL(rp1&+4) :vp1&=scr1&+44
WINDOW CLOSE 2 :tt-="Bitte warten"
t-="Tempo = +/-     beliebige Taste = ende"+CHR-(0)
t1&=SADD(t-):SCREEN 2,sw,sh,tiefe,1: WINDOW 3,tt-,,0,2
scr2&=PEEKL(WINDOW(7)+46)
rp2&=scr2&+84 :mb2&=PEEKL(rp2&+4) :vp2&=scr2&+44
RETURN

Farben:
FOR i=0 TO 7:READ f(i):NEXT
CALL LoadRGB4&(vp2&,VARPTR(f(0)),8)
CALL LoadRGB4&(vp1&,VARPTR(f(0)),8)
RETURN

Bildeinlesen:
fina-=":Bild/MrBrown"+CHR-(0)
offen&=xOpen&(SADD(fina-),1005)
IF offen&=0 THEN fehl=-1:RETURN
lese&=xRead&(offen&,adr&,12)
suchen=-1
WHILE suchen
  lese&=xRead&(offen&,adr&,8)
  chlg&=PEEKL(adr&+4)
  IF chlg&/2-FIX(chlg&/2)>0 THEN chlg&=chlg&+1
  text-="":FOR i=0 TO 3
  text-=text-+CHR-(PEEK(adr&+i)):NEXT
  IF text-="BODY" THEN suchen=0:GOTO wtr
  chunk%=chlg&
  rest&=xRead&(offen&,adr&,chunk%)
  wtr:
WEND
bbreite=79 :bhoehe=84 :bytprozeile=bbreite/8
ro=11:rl=4    'Rand oben und links
bitmap1&=mb2&+8 :diff=120/8+(70*320/8)
FOR h= 0 TO bhoehe-1
  FOR eb=0 TO tiefe-1
    zeile&=PEEKL(bitmap1&+(4*eb))+40*h+diff
    lese&=xRead&(offen&,zeile&,bytprozeile)
NEXT eb,h
```

```
IF offen& THEN CALL xClose&(offen&)
RETURN

BilderZeichnen:
h=72 :b=72 :kx=122 :ky=72 :diff=14
CALL SetAPen&(rp1&,0)      'Screen-Titelzeile loeschen
CALL RectFill&(rp1&,0,0,319,10)
CALL SetAPen&(rp1&,1)
pi!=3.141593:bo!=pi!/180
w!=SIN(45*bo!)       'gleicher Winkel bei cos 45 Grad
GOSUB Grad0          '0 Grad konstruieren
GOSUB Grad45         '45 Grad konstruieren
GOSUB Grad90         '90 Grad konstruieren
GOSUB Grad135        '135 Grad konstruieren
GOSUB Grad180        '180 Grad konstruieren
GOSUB Grad225        '225 Grad konstruieren
GOSUB Grad270        '270 Grad konstruieren
GOSUB Grad315        '315 Grad konstruieren
GOSUB Grad0          '0 Grad konstruieren
CALL SetWindowTitles&(WINDOW(7),t1&,-1)
RETURN

Grad0:
di=4          'Bild zentrieren
CALL SetAPen&(rp1&,0)      'Puffer 1 loeschen
CALL RectFill&(rp1&,0,0,79,79)
CALL SetAPen&(rp1&,1)
'Bild auf Screen1 Position 1 uebertragen
CALL ClipBlit&(rp2&,122,72,rp1&,di,0,b-di,h,192)
RETURN

Grad45:
b1=h/2:h1=(101-80)/2
FOR ho=0 TO h-1
  FOR br=0 TO b-1
    col=ReadPixel&(rp1&,br,ho)
    CALL SetAPen&(rp1&,col)
    asx=br*w!-ho*w!+80+b1
    asy=br*w!+ho*w!
    erg&=WritePixel&(rp1&,asx,asy)
    asx=(br+.5)*w!-(ho+.5)*w!+80+b1
    asy=(br+.5)*w!+(ho+.5)*w!
```

```
    erg&=WritePixel&(rp1&,asx,asy)
  NEXT br
  CALL moveScreen&(scr2&,Ø,2)
NEXT ho
CALL ClipBlit&(rp1&,8Ø,h1,rp1&,8Ø,Ø,b,h+1Ø,192)
RETURN

Grad9Ø:
'Bild von Position 1 drehen, an Position 3
FOR i=Ø TO b-1
  FOR j=Ø TO h-1
  CALL ClipBlit&(rp1&,i,j,rp1&,159+b-j,i,1,1,192)
NEXT j,i
RETURN

Grad135:
'Bild von Position 2 drehen, an Position 4
FOR i=Ø TO b-1
  FOR j=Ø TO h-1
  CALL ClipBlit&(rp1&,i+8Ø,j,rp1&,239+b-j,i,1,1,192)
NEXT j,i
RETURN

Grad18Ø:
'Spiegeln GradØ um X-Achse in Puffer 8
FOR i=Ø TO h-1
  CALL ClipBlit&(rp1&,Ø,i,rp1&,24Ø,79+h-i,b,1,192)
NEXT
'seitenverkehrten Puffer8 spiegeln in Bild 5
FOR i=Ø TO b-1
  CALL ClipBlit&(rp1&,24Ø+i,8Ø,rp1&,b-1-i,8Ø,1,h,192)
NEXT
RETURN

Grad225:
'Spiegeln Grad45 um X-Achse in Puffer 8
FOR i=Ø TO h-1
  CALL ClipBlit&(rp1&,8Ø,i,rp1&,24Ø,79+h-i,b,1,192)
NEXT
'seitenverkehrten Puffer8 spiegeln in Bild 6
FOR i=Ø TO b-1
  CALL ClipBlit&(rp1&,24Ø+i,8Ø,rp1&,79+b-i,8Ø,1,h,192)
NEXT
RETURN
```

```
Grad27Ø:
'Spiegeln Grad9Ø um X-Achse in Puffer 8
FOR i=Ø TO h-1
  CALL ClipBlit&(rp1&,16Ø,i,rp1&,24Ø,79+h-i,b,1,192)
NEXT
'seitenverkehrten Puffer 8 spiegeln in Bild 7
FOR i=Ø TO b-1
  CALL ClipBlit&(rp1&,24Ø+i,8Ø,rp1&,159+b-i,8Ø,1,h,192)
NEXT
RETURN

Grad315:
'Spiegeln Grad135 um X-Achse in Puffer 1
FOR i=Ø TO h-1
  CALL ClipBlit&(rp1&,24Ø,i,rp1&,Ø,h-1-i,b,1,192)
NEXT
'seitenverkehrten Puffer 1 spiegeln in Bild 8
FOR i=Ø TO b-1
  CALL ClipBlit&(rp1&,i,Ø,rp1&,239+b-i,8Ø,1,h,192)
  CALL moveScreen&(scr2&,Ø,-2)
NEXT i
RETURN

SUB Taste STATIC
  SHARED a%
  ta-=INKEY-
  IF ta-<>"" THEN
    a%=Ø
    IF ta-=CHR-(43) THEN
      tempo%=tempo%-1ØØ :a%=-1
      IF tempo%<Ø THEN tempo%=Ø
    END IF
    IF ta-=CHR-(45) THEN
      tempo%=tempo%+1ØØ :a%=-1
      IF tempo%>1ØØØØ THEN tempo%=1ØØØØ
    END IF
  END IF
  FOR i = Ø TO tempo%:NEXT
END SUB
```

```
SUB Bild STATIC
  SHARED rp1&,rp2&
  x%=x%+8Ø:IF x%>=32Ø THEN x%=Ø: y%=y%+8Ø
  IF y%>=16Ø THEN y%=Ø
  CALL ClipBlit&(rp1&,x%,y%,rp2&,122,72,72,72,192)
END SUB

Farbwerte:
DATA &HE,&HFØØ,&hØfbØ,&hØf9Ø
DATA &hØe9Ø,&hØc9Ø,&hbØ,&hØØØ
```

Die beiden Subroutinen »LibraryOeffnen« und »SpeicherReservieren« zeigen nichts wesentlich Neues. In »SchirmFenster« öffnen wir zwei neue Bildschirme. Dabei ist SCREEN 1 als Puffer vorgesehen. Da er die 8 Grafiken (siehe Bild) aufnehmen wird, genügt eine Höhe von 160 Bildpunkten. SCREEN 2 ist der sichtbare Bildschirm. Für ihn wird das Fenster 3 mit dem Fenster-Text »Bitte warten« eingerichtet. Die benötigten Speicherstellen für die RastPorts, ViewPorts etc. werden in Variablen festgehalten.

In der Subroutine »Farben« werden für die beiden ViewPorts die gleichen Farben eingelesen und gesetzt. Normalerweise ist das für den Screen, der im Hintergrund bleibt, nicht nötig. Wenn Sie das Programm schon gestartet haben, wissen Sie auch den Grund dafür. Anschließend wird das Bild *Mr. Brown* eingelesen. Auch hier können Sie eine andere Grafik einlesen lassen. Ideal wäre eine Grafik mit 72 * 72 Bildpunkten Größe. Eventuell müssen Sie die Grafik in Bild 1 anders zentrieren.

Bei »BilderZeichnen« werden in den Puffer-Bildschirm 1 die Grafiken gezeichnet. Da dabei die Titelleiste des Screens stört, löschen wir diese mit einem Rechteck (*RectFill*) in der Hintergrundfarbe. Für die Konstruktion der 45-Grad-Grafik errechnen wir den SIN(45) und speichern das Ergebnis in *w!*. Da bei 45 Grad SIN und COS den gleichen Wert erhalten, genügt *w!* für beide Funktionen. Das Zeichnen der Grafiken selbst haben wir bereits im letzten Kapitel besprochen. Damit der Anwender nicht 2 Minuten und 40 Sekunden auf den Beginn des Programmes warten muß, ist in zwei Subroutinen ein kleiner Gag eingebaut. Bei »Grad45« finden Sie in der äußeren Schleife die Library-Routine *MoveScreen*. Bei jedem Schleifendurchlauf wird der vordere Bildschirm um zwei Pixel nach unten geschoben. Dadurch kann der Anwender einen Blick hinter die Kulissen werfen und verfolgen, wie die einzelnen Grafiken gezeichnet werden. Wird das letzte Bild bei »Grad315« gezeichnet, wird der Bildschirm, ebenfalls in einer Schleife, wieder nach oben geschoben. Mit *SetWindowTitles* wird durch einen neuen Fenster-Text angezeigt, daß die Warterei ein Ende hat.

Die Drehung des *Mr. Brown* läuft in der WHILE/WEND-Schleife ab. Bei jedem Schleifendurchlauf werden die beiden Unterprogramme »Bild« und »Taste« aufgerufen. Bei »Bild« werden die Variablen *x%* und *y%* auf das nächste Feld des Puffer-Screens

gesetzt und die Grafik vom Screen 1 in den Screen 2 kopiert. Das Unterprogramm hatte ursprünglich folgendes Aussehen:

```
SUB Bild STATIC
  SHARED rp1&,rp2&,kx%,ky%,b%,h%
  x%=x%+80:IF x%>=320 THEN x%=0: y%=y%+80
  IF y%>=160 THEN y%=0
  CALL ClipBlit&(rp1&,x%,y%,rp2&,kx%,ky%,b%,h%,192)
END SUB
```

Dadurch könnte das Unterprogramm für verschiedene Bildgrößen (solange sie in den Zwischenspeicher passen) und unterschiedliche Positionen auf dem Bildschirm aufgerufen werden. Diese ordentlichere Programmierung bringt aber einen gewaltigen Zeitnachteil und reduziert die Anzahl der kopierten Bilder beträchtlich. Um es besser auszudrücken, es benötigt, um die gleiche Anzahl von Bildern zu kopieren, 75% (!!!) mehr Zeit als die abgemagerte Variante des Programmes. Sie sehen an diesem Beispiel, daß es sich immer lohnt, ein Programm nach eventuellen Hemmschuhen abzuklopfen.

Das zweite Unterprogramm fragt die Tasten *plus* und *minus* ab und erhöht oder reduziert je nach gedrückter Taste eine Zeitschleife. Mit einem Druck auf irgendeine andere Taste wird die Schleife verlassen, und man gelangt zum Label »ende«. Dort wird das Programm ordnungsgemäß beendet.

8.5 Flächen außer Rand und Band

Sicherlich haben Sie schon in professionellen Programmen Grafiken oder Texte, die beim Amiga auch Grafik sind, gesehen, die kunstvoll verformt auf den Bildschirm gebracht wurden. Interessante und eindrucksvolle Effekte lassen sich dadurch realisieren. Mit bestimmten Verformungen läßt sich auch der Eindruck einer dritten Dimension erwecken. Ein Grafikbild könnte zum Beispiel um eine Litfaßsäule gezeichnet werden.

8.5.1 Verzerren im Detail

Mit den inzwischen erworbenen Kenntnissen über die Transformationen von Pixel-Grafiken sollten die geschilderten Effekte keine große Herausforderung mehr darstellen. Die ganze Kunst liegt darin, eine Grafik zeilen- oder spaltenweise zu verschieben. Dazu verpassen wir der Grafik, bzw. dem rechteckigen Bildschirmausschnitt der Grafik, wieder ein eigenes Koordinatenkreuz. Nun braucht man nur entlang der X- oder der Y- Achse die einzelnen Zeilen oder Spalten nach einer beliebigen Rechen-Funktion zu verschieben. Denkbar ist auch eine gleichzeitige Verschiebung um beide Achsen. Da wir oder ein Anwender nie zuviel Zeit haben, soll das Ganze sehr schnell vonstatten gehen. Die schnellen Grafik-Routinen werden uns dabei unterstützen.

Welche Funktion Sie wann einsetzen können, bleibt Ihrem Geschmack vorbehalten. Probieren geht auch hier über Studieren. Um einen besseren Eindruck von der geschilderten Technik zu erhalten, betrachten wir uns die Verzerrung entlang einer Sinus-Kurve. Zur übersichtlicheren Darstellung verzichten wir auf einen Puffer. Das zu verändernde Bild und die transformierte Grafik befinden sich also im gleichen Bildschirm.

Lage der ursprünglichen Grafik (o.l.):	bx=10:by=20
Lage der geänderten Grafik (o.l.):	bx2=150:by=20
Breite der ursprünglichen Grafik:	b1=65

Die Verzerrung soll entlang der X-Achse erfolgen. Die Winkel bei den Amiga-Winkelfunktionen müssen im Bogenmaß angegeben werden. Der maximale Winkel von 360 Grad entspricht einem Bogenmaß von 2*pi. Damit die Schwingung der Bildbreite angeglichen werden kann, errechnen wir, wieviel Grad der Breite eines Bildpunktes bzw. einer Grafikspalte entsprechen:

```
v!=2*pi!/b1
```

Damit die Schwingung erkennbar wird, legen wir in der Variablen *vergr* einen Vergrößerungsfaktor von 15 fest. Nun können wir für jede Spalte der X-Achse den Ausschlag der Sinuskurve nach der Funktion y=SIN(x) berechnen.

```
Y-Schwingung(SpaltenpositionX)=SIN(SpaltenpositionX*v!)*vergr
```

Die Berechnung der Funktion packen wir zusammen mit der Kopier-Routine *ClipBlit* in eine Schleife, die die Bildbreite heraufzählt und schon ist die verzerrte Grafik fertig:

```
FOR x= Ø TO b1
  ys(x)=by+vergr*SIN(x*v!)
  CALL ClipBlit&(rp&,bx+x,by,rp&,x+bx2,ys(x),1,h1,192)
NEXT
```

Damit das Bild schnell genug gezeichnet werden kann (0,3 bis 0,4 Sekunden), brauchen wir nur noch die verschiedenen Y-Werte der Schwingung vor dem Zeichnen berechnen und in einer Feldvariablen speichern.

8.5.2 Deform, verformen im Sekundenbruchteil

In dem folgenden Programmbeispiel sind neben der besprochenen Funktion fünf weitere Funktionen eingesetzt. Das Programm ist so übersichtlich aufgebaut, daß Sie Ihre eigenen Vorstellungen und Wünsche leicht mit einbauen können.

```
REM deform  Pfad: Darstellung/8TransPixel/deform
'P8-5
'Dauer einer Verformung ca. Ø,4 sec
```

```
CLEAR
DEFINT a-z
b1=65:DIM ys(b1),yc(b1),yk(b1)
h1=73:DIM xs(h1),xc(h1),xk(h1)
GOSUB LibraryOeffnen
GOSUB SpeicherReservieren :IF fehl THEN Fehler2
GOSUB SchirmFenster
GOSUB farben
GOSUB Bildeinlesen :IF fehl THEN Fehler1
GOSUB VariablenFelder

start:
ta-=INKEY-:IF ta-<CHR-(129) OR ta->CHR-(135) THEN start
wahl= ASC(ta-)-128
LINE (115,Ø)-(WINDOW(2),WINDOW(3)),Ø,bf
ON wahl GOTO sinX,sinY,cosX,cosY,KreisX,KreisY,ende

ende:
IF adr& THEN CALL FreeMem&(adr&,8ØØ)
WINDOW CLOSE 2 :SCREEN CLOSE 2
IF fehl THEN
  BEEP:PRINT ft-:t&=TIMER:WHILE t&+5>TIMER:WEND
END IF
ERASE xs,ys,xc,yc,xk,yk
LIBRARY CLOSE :END

Fehler1:ft-= "Fehler beim Einlesen des Bildes":fehl=-1:GOTO ende
Fehler2:ft-= "Speicher nicht ausreichend":fehl=-1:GOTO ende

LibraryOeffnen:
DECLARE FUNCTION AllocMem&() LIBRARY
DECLARE FUNCTION xOpen&() LIBRARY
DECLARE FUNCTION xRead&() LIBRARY
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/dos.library"
LIBRARY ":bue/exec.library"
RETURN

SpeicherReservieren:
adr&=AllocMem&(8ØØ,65539&):IF adr&=Ø THEN fehl=-1
RETURN
```

```
SchirmFenster:
SCREEN 2,32Ø,2ØØ,3,1: WINDOW 2,,(Ø,Ø)-(31Ø,18Ø),Ø,2
scr&=PEEKL(WINDOW(7)+46)
rp&=scr&+84 :mb&=PEEKL(rp&+4) :vp&=scr&+44
RETURN

farben: FOR i=Ø TO 7:READ f(i):NEXT
CALL LoadRGB4&(vp&,VARPTR(f(Ø)),8)
RETURN

Bildeinlesen:
fina-=":Bild/MrBrown"+CHR-(Ø):offen&=xOpen&(SADD(fina-),1ØØ5)
IF offen&=Ø THEN fehl=-1
lese&=xRead&(offen&,adr&,12):suchen=-1:tiefe=3
WHILE suchen
  lese&=xRead&(offen&,adr&,8):chlg&=PEEKL(adr&+4)
  IF chlg&/2-FIX(chlg&/2)>Ø THEN chlg&=chlg&+1
  text-="":FOR i=Ø TO 3
  text-=text-+CHR-(PEEK(adr&+i)):NEXT
  IF text-="BODY" THEN suchen=Ø:GOTO wtr
  chunk%=chlg&:rest&=xRead&(offen&,adr&,chunk%)
  wtr:
WEND
bbreite=79 :bhoehe=84 :bytprozeile=bbreite/8
bitmap1&=mb&+8 :diff=12Ø/8+(7Ø*32Ø/8)
FOR h= Ø TO bhoehe-1: FOR eb=Ø TO tiefe-1
    zeile&=PEEKL(bitmap1&+(4*eb))+4Ø*h+diff
    lese&=xRead&(offen&,zeile&,bytprozeile)
NEXT eb,h
IF offen& THEN CALL xClose&(offen&)
RETURN

VariablenFelder:
pi!=3.141593 :ob=72 :li=122:vergr=15
bx=1Ø:bx2=15Ø:by=2Ø
CALL ClipBlit&(rp&,li,ob,rp&,bx,by,b1,h1,192)
LOCATE 14,1:PRINT "F1: y=sin(x)":GOSUB feldsinX
LOCATE 15,1:PRINT "F2: x=sin(y)":GOSUB feldsinY
LOCATE 16,1:PRINT "F3: y=cos(x)":GOSUB feldcosX
LOCATE 17,1:PRINT "F4: x=cos(y)":GOSUB feldcosY
LOCATE 18,1:PRINT "F5: y=Kreis(x)":GOSUB feldKreisX
LOCATE 19,1:PRINT "F6: x=Kreis(y)":GOSUB feldKreisY
```

```
LOCATE 2Ø,1:PRINT "F7: E N D E"
RETURN

feldsinX:     'Winkelfunktion y=sin(x)
v!=2*pi!/b1
FOR x= Ø TO b1:ys(x)=by+vergr*SIN(x*v!):NEXT :RETURN
sinX:
FOR x= Ø TO b1
  CALL ClipBlit&(rp&,bx+x,by,rp&,x+bx2,ys(x),1,h1,192)
NEXT :GOTO start

feldsinY:     'Winkelfunktion x=sin(y)
v!=2*pi!/h1
FOR y= Ø TO h1:xs(y)=bx2+vergr*SIN(y*v!):NEXT :RETURN
sinY:
FOR y= Ø TO h1
  CALL ClipBlit&(rp&,bx,by+y,rp&,xs(y),y+by,b1,1,192)
NEXT :GOTO start

feldcosX:    'Winkelfunktion y=cos(x)
v!=2*pi!/b1
FOR x= Ø TO b1:yc(x)=by+vergr*COS(x*v!):NEXT :RETURN
cosX:
FOR x= Ø TO b1
  CALL ClipBlit&(rp&,bx+x,by,rp&,x+bx2,yc(x),1,h1,192)
NEXT :GOTO start

feldcosY:     'Winkelfunktion x=cos(y)
v!=2*pi!/h1
FOR y= Ø TO h1:xc(y)=bx2+vergr*COS(y*v!):NEXT :RETURN
cosY:
FOR y= Ø TO h1
  CALL ClipBlit&(rp&,bx,by+y,rp&,xc(y),y+by,b1,1,192)
NEXT :GOTO start

feldKreisX:     'Hoehensatz y=sqr(x*(c-x))
FOR x= Ø TO b1:yk(x)=by+SQR(x*(b1-x)):NEXT :RETURN
KreisX:
FOR x= Ø TO b1
  CALL ClipBlit&(rp&,bx+x,by,rp&,x+bx2,yk(x),1,h1,192)
NEXT :GOTO start
feldKreisY:     'Hoehensatz x=sqr(y*(c-y))
FOR y= Ø TO h1:xk(y)=bx2+SQR(y*(h1-y)):NEXT :RETURN
```

```
KreisY:
FOR y= Ø TO h1
  CALL ClipBlit&(rp&,bx,by+y,rp&,xk(y),y+by,b1,1,192)
NEXT :GOTO start

Farbwerte:
DATA &HE,&HFØØ,&hØfbØ,&hØf9Ø
DATA &hØe9Ø,&hØc9Ø,&hbØ,&hØØØ
```

Zu Beginn werden gleich die Variablenfelder für die 6 Funktionen dimensioniert. Bei eigenen Programmerweiterungen ergänzen Sie die Dimensionierung entsprechend. Es folgen die Subroutinen zum Öffnen der Library, zur Speicherreservierung, zum Öffnen eines neuen Bildschirmes mit Fenster, zur Eingabe einer neuen Farbtafel und zum Einlesen einer Grafik.

Die Subroutine »VariablenFelder« schreibt die entsprechenden Funktionen ins Fenster und verzweigt weiter zu den einzelnen Berechnungen der Funktionswerte. Bei Ergänzungen haben Sie, ohne die Textausgabe verändern zu müssen, noch Platz für 3 Einträge. Bei den einzelnen Verzweigungen sind, zur besseren Übersicht, immer die Subroutinen für die Berechnung der Variablen-Felder und zum Kopieren der Grafik zusammengefaßt.

Zum Programmablauf selbst ist nicht mehr viel zu sagen. Beim Label »start« wartet das Programm auf die Betätigung einer Funktionstaste. Mit einer Block-Anweisung wird die alte Grafik gelöscht und je nach gedrückter Taste zur Kopier-Routine verzweigt. Ist die Grafik fertiggezeichnet, verzweigt das Programm zurück zur Sprungmarke »start«.

Sie haben an diesem Programm gesehen, daß mit dem Basic des Amiga solche Effekte problemlos und vor allem auch schnell zu programmieren sind.

Kapitel 9

3-D-Grafik

Wie wir es auch drehen und wenden; real sitzen wir vor einem Bildschirm, der nur zwei Dimensionen sein eigen nennt. Die Kunst der dreidimensionalen Darstellung besteht also darin, die Illusion einer dritten Dimension zu erwecken. Ein einfacher Schatten an einem grafischen Objekt kann schon einen gewissen Eindruck von Tiefe darstellen. Bessere Techniken bringen auch bessere Resultate. Je wirklichkeitsgetreuer die Grafik gezeichnet wird, um so mehr steigt der Programmieraufwand und die zur Zeichnung des Bildes benötigte Zeit. Aber auch dann muß man sich auf berechenbare geometrische Objekte stützen. Sie merken schon, daß es sich bei der dreidimensionalen Grafik um ein sehr komplexes Thema handelt. Bereits Teilbereiche aus diesem Themen-Komplex sind nur in einem eigenen Buch abzuhandeln. Sie finden daher in diesem Kapitel nur einen Anriß zu dem großen Thema 3 D. Wenn Sie Geschmack daran finden, steht es Ihnen offen, Ihr Wissen weiter zu vertiefen.

9.1 Start in die dritte Dimension

Achsenkreuze und Koordinatensysteme haben Sie bereits in den letzten Kapiteln kennengelernt. Gegenüber der zweidimensionalen Darstellung haben Sie es nun mit einer dritten Achse z zu tun. Entgegen den Gepflogenheiten mancher Mathematikbücher lassen wir die X- und die Y-Achse dort, wo wir es gewohnt sind. In Bild 9.1 ist ein Grafikpunkt in einem räumlichen, kartesischen Koordinatensystem dargestellt.

Bei der zweidimensionalen Grafik können wir einen Punkt mit den Koordinaten x und y beschreiben P(x,y). Bei der dreidimensionalen Darstellung wird dann konsequenterweise ein Punkt durch drei Koordinaten festgelegt P(x,y,z). In unserer kleinen Skizze (siehe Seite 194) ragt die Z-Achse nach hinten ins Bild hinein. Man nennt das ein Linkssystem. Dabei werden die Werte für z negativ. Im Gegensatz dazu gibt es noch das Rechtssystem, bei dem die Z-Achse nach vorne zeigt. Wir bleiben aber beim leichter darstellbaren Linkssystem.

Das Koordinatensystem aus unserer Skizze nennen wir Weltsystem, weil es für die einzelnen Grafikpunkte im System eine eigene Welt darstellt. Alle Elemente in dieser Welt

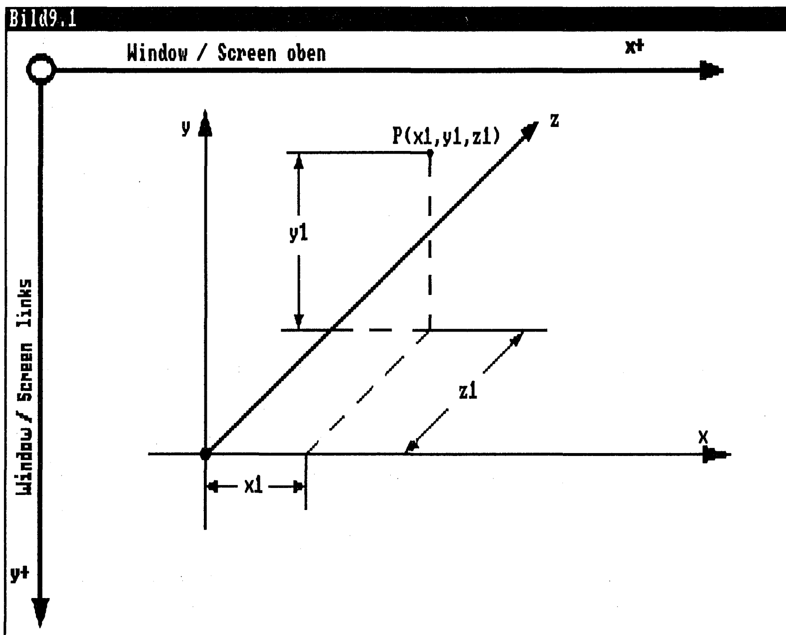


Bild 9.1: Ein Punkt in einem räumlichen Koordinatensystem

lassen sich durch das Koordinatensystem beschreiben. Dabei kann der Nullpunkt des Koordinatenkreuzes irgendwo im Raum liegen. Zu diesem Weltsystem benötigen wir aber mindestens noch ein zweites System. Ich spreche von dem System unseres Bildschirmes. Dieses System ist zweidimensional und hat seinen Nullpunkt in der linken oberen Ecke.

9.2 Projektion und Perspektive

Irgendwie muß man das räumliche Gebilde des Weltsystems wieder in das zweidimensionale Bildschirm-System praktizieren können. Am leichtesten kann man sich das anhand des Prinzips einer Fernsehkamera (Fotoapparat, Filmkamera, Schattenwurf) vorstellen. Sie nimmt die dreidimensionale Realität auf und liefert uns das zweidimensionale Fernsehbild. Dabei werden die Gegenstände, von einem zentral im Hintergrund liegenden Punkt ausgehend, auf den Bildschirm projeziert. Man nennt dieses Prinzip die Zentralprojektion. Dabei werden die weiter entfernt liegenden Gegenstände kleiner dargestellt als die Objekte, die näher an der Kamera sind. Das gibt dann zum Beispiel das Bild einer Eisenbahnschiene, die sich immer mehr verjüngt, bis nur noch ein Punkt im Hintergrund zu erkennen ist. Diese Art der Darstellung läßt sich auch in der Computer-Grafik realisieren. Allerdings sind dafür aufwendige Berechnungen notwendig.

Eine andere Möglichkeit ist die, die Strahlen nicht von einem zentralen, entfernt liegenden Punkt auszusenden, sondern sie parallel zu den einzelnen Kanten der Objekte bis zum Bildschirm laufen zu lassen. Wegen der parallel laufenden Strahlen nennt man dieses Prinzip *Parallelprojektion*. Da dieses Prinzip sich mit geringem Rechenaufwand verwirklichen läßt, werden wir es in unseren Beispielen einsetzen.

Um einen Gegenstand auch als Körper wahrnehmen zu können, müssen wir ihn unter einem bestimmten Blickwinkel betrachten. So ist ein Würfel, den wir direkt von vorne betrachten, nicht als Würfel zu erkennen, sondern nur als quadratische Fläche. Um einen Blickwinkel zur perspektivischen Darstellung zu erreichen, bewegen wir uns natürlich nicht selbst (was bei einem zweidimensionalen Bildschirm wohl kaum einen Sinn ergibt), sondern drehen das Weltsystem. Dabei können wir die Welt mit ihren Objekten um alle drei Achsen drehen. Wir begnügen uns aber mit Drehungen um die X- und Y-Achse. Dabei bedeutet eine Drehung um die Y-Achse eine Veränderung des horizontalen Blickwinkels. Wir sehen das Objekt also von links oder rechts. Dagegen zeigt eine Drehung um die X-Achse eine vertikale Änderung des Blickwinkels. Wir sehen also das Objekt von oben oder von unten.

Bei der Parallelprojektion kann die Umrechnung der drei Koordinatenwerte eines Punktes P(x,y,z) im Raum in die beiden Koordinatenwerte P(x,y) der Bildschirmebene nach folgenden Formeln erfolgen:

```
xe=xk+zr*SIN(yw)+xr*COS(yw)
ye=yk-(yr*COS(xw)+xr*SIN(xw)*SIN(yw)-zr*SIN(xw)*COS(xw))
```

Die einzelnen Parameter haben folgende Bedeutung:

- xe – Koordinate x des Punktes in der Ebene.
- ye – Koordinate y des Punktes in der Ebene.
- xk – Abstand x des Koordinatenkreuzes der Ebene von der linken Bildschirmkante.
- yk – Abstand y des Koordinatenkreuzes der Ebene von der oberen Bildschirmkante.
- xr – Koordinate x des Punktes im Raum.
- yr – Koordinate y des Punktes im Raum.
- zr – Koordinate z des Punktes im Raum.
- xw – Drehwinkel um die X-Achse.
- yw – Drehwinkel um die Y-Achse.

Mit diesen beiden Formeln bestens gerüstet, können wir uns an unser erstes 3-D-Programm wagen. Wir wollen eine Pyramide mit quadratischer Grundfläche in Paral-

lelprojektion zeichnen. Das zugehörige Koordinatenkreuz soll um die X- und Y-Achse gedreht werden können. Zur Drehung sollen die Cursor-Tasten herangezogen werden.

```
REM Pyramide  Pfad: Darstellung/9dreiD.Grafik/Pyramide
'P9-1
'Parallelprojektion

DEFINT a-z
n=21          'Anzahl der Punkte
DIM xr(n),yr(n),zr(n),xe(n),ye(n),wc!(359),ws!(359)

'Anweisung an den Anwender
  PALETTE Ø,.6,.6,.6
  PRINT "Bitte Cursortasten"
  PRINT "Drehung Y-Achse <-- und -->"
  PRINT "Drehung X-Achse  ^  und  \/"
  PRINT :PRINT "ENDE = RETURN"

'Berechnung aller Winkel sin und cos
  pi!=3.141593
  bog!=pi!/18Ø
  FOR i = Ø TO 359
    wc!(i)=COS(i*bog!)
    ws!(i)=SIN(i*bog!)
  NEXT

xk=4ØØ:yk=1ØØ                'Nullpunkt Koordinatenkreuz
breite=WINDOW(2)-1
hoehe =WINDOW(3)-1
'Raumkoordinaten einlesen
'fuer Koordinatenkreuz und Pyramide
'xr() yr() zr() Koordinaten x,y und z im Raum
  FOR i = Ø TO n
    READ xr(i),yr(i),zr(i)
    yr(i)=yr(i)/2            'wegen Pixelhoehe x/y
  NEXT

'Startposition
  xw=2Ø:yw=33Ø               'Startwinkel
  GOSUB zeichnen
a=-1
WHILE a
  ta-=INKEY-
  IF ta->CHR-(27) AND ta-<CHR-(32) THEN
```

```
    IF ta-=CHR-(28) THEN xw=xw-1:IF xw<Ø THEN xw=359 'Cursor oben
    IF ta-=CHR-(29) THEN xw=xw+1:IF xw>359 THEN xw=Ø 'unten
    IF ta-=CHR-(3Ø) THEN yw=yw-1:IF yw<Ø THEN yw=359 'rechts
    IF ta-=CHR-(31) THEN yw=yw+1:IF yw>359 THEN yw=Ø 'links
    GOSUB zeichnen
  END IF
  IF ta-=CHR-(13) THEN a=Ø
WEND
END

zeichnen:
  FOR i = Ø TO n
    xe(i)=xk+zr(i)*ws!(yw)+xr(i)*wc!(yw)
    ye(i)=yk-(yr(i)*wc!(xw)+xr(i)*ws!(xw)*ws!(yw)-
**                                 zr(i)*ws!(xw)*wc!(yw))
  NEXT i
  LINE (24Ø,Ø)-(breite,hoehe),Ø,bf
  FOR i = 6 TO n-1 STEP 2
    LINE (xe(i),ye(i))-(xe(i+1),ye(i+1)),1
  NEXT
  fa=1
  FOR i = Ø TO 4 STEP 2
    LINE (xe(i),ye(i))-(xe(i+1),ye(i+1)),fa
    fa=fa+1
  NEXT
RETURN

Koordinatenkreuz:
DATA Ø,Ø,Ø,  16Ø,Ø,Ø
DATA Ø,Ø,Ø,  Ø,16Ø,Ø
DATA Ø,Ø,Ø,  Ø,Ø,-16Ø

Pyramide:
DATA Ø,Ø,Ø,        1ØØ,Ø,Ø
DATA 1ØØ,Ø,Ø,      1ØØ,Ø,-1ØØ
DATA 1ØØ,Ø,-1ØØ,   Ø,Ø,-1ØØ
DATA Ø,Ø,-1ØØ,     Ø,Ø,Ø
DATA Ø,Ø,Ø,        5Ø,14Ø,-5Ø
DATA 5Ø,14Ø,-5Ø,   1ØØ,Ø,-1ØØ
DATA 5Ø,14Ø,-5Ø,   1ØØ,Ø,Ø
DATA 5Ø,14Ø,-5Ø,   Ø,Ø,-1ØØ
```

Zuerst dimensionieren wir die Koordinaten der einzelnen Eckpunkte der Pyramide und des Koordinatenkreuzes im Raum und in der Ebene. Auch sämtliche Sinus- und Cosinus-Winkelwerte werden dimensioniert. Diese berechnen wir gleich anschließend und speichern sie in den beiden Feldern. Damit ersparen wir uns bei der Berechnung der Koordinaten unnötige Rechenzeit. Nun lesen wir die Raumkoordinaten der Eckpunkte aus den Datenwerten ein. Dabei sind immer zwei Eckpunkte zusammengefaßt, die später mit der LINE-Anweisung gezeichnet werden. Durch diese Methode der Datenspeicherung kommen etliche Punkte mehrfach vor. Die Y-Werte werden wegen des Pixelverhältnisses X/Y halbiert, da wir uns im Hires-Modus befinden.

Damit kann bereits das Startbild gezeichnet werden. Das Hauptprogramm läuft in der WHILE/WEND-Schleife. In der Schleife werden die Cursortasten abgefragt. Je nach der betätigten Taste werden die Drehwinkel *xw* und *yw* um ein Grad erhöht oder vermindert. Nach jeder Änderung wird die Grafik neu gezeichnet. Ein Tastendruck auf RETURN beendet das Programm.

Kommen wir nun zum wichtigsten Programmteil, zur Subroutine »zeichnen«. In der ersten Schleife rechnen wir die drei Raumkoordinaten in die beiden Ebenenkoordinaten der Punkte um. Die beiden Formeln haben wir uns bereits vorher angeschaut. Ein LINE-Befehl mit der Option *bf* löscht die bisherige Grafik. In der folgenden FOR/NEXT-Schleife werden die einzelnen Linien der Pyramide auf den Bildschirm gebracht. Zum Schluß wird das Koordinatenkreuz gezeichnet. Dabei wird die X-Koordinate weiß, die Y-Koordinate schwarz und die Z-Koordinate orange gezeichnet. Für eigene Versuche ändern Sie einfach die DATAs entsprechend Ihrer eigenen Grafik ab.

9.3 Verschieben und Drehen im Raum

Mit den zweidimensionalen Grafiken haben wir fast alle Möglichkeiten der Transformation von Grafiken realisiert. In diesem Kapitel versuchen wir, ob das mit der dreidimensionalen Grafik auch so schön klappt. Sie werden staunen, was Sie mit relativ einfachen Mitteln alles zustande bringen können.

9.3.1 Rotierendes Drahtmodell

Nachdem wir in den beiden letzten Kapiteln einige Grundlagen über die dreidimensionale Grafik erarbeitet haben, sollte es uns nicht besonders schwerfallen, die Kenntnisse in einem praktischen Beispiel einzusetzen. Damit wir wissen, was wir eigentlich programmieren wollen, stellen wir zuerst den Anforderungskatalog zusammen. Für den Anfang soll es ein einfaches grafisches Objekt sein. Wegen der einfachen Berechnung drängt sich dafür der Würfel geradezu auf. Besonders eindrucksvoll ist die Drehung von Körpern. Also wird sich unser Würfel drehen. Die Drehung soll möglichst flott,

aber ohne zu rucken vonstatten gehen. Damit der Anwender das Programm beeinflussen kann, wird die Größe des Würfels, innerhalb der Fenstergrenzen, frei wählbar sein.

Nun können wir uns an die Verwirklichung des Projektes wagen. Als Bildschirm wählen wir einen Screen mit niedriger Auflösung. Damit können wir das Pixelverhältnis x/y vernachlässigen. Für eine Liniengrafik kommen wir mit einer Farbe aus. Uns genügt daher die Tiefe 1. Die Größe des Würfels holen wir uns über eine INPUT-Anweisung. Die maximale Größe richtet sich nach der Höhe des Screens (200/256). Der nächste Punkt ist die Frage nach der Lage des Koordinatenkreuzes. Wegen der einfacheren Berechnung bietet sich dafür die Mitte des Würfels an. Damit können wir den Nullpunkt des Koordinatenkreuzes auch in die Mitte des Bildschirmes legen.

Damit sind wir bereits bei der Festlegung der Eckpunkte des Würfels im Raum. Da wir den Nullpunkt in die Mitte des Würfels gelegt haben, sind alle Maße der Punkte für *x*, *y* und *z* eine halbe Kantenlänge vom Mittelpunkt entfernt. Mit diesen Werten können wir uns an die Umrechnung der Raumpunkte in die Ebenenpunkte mit der Technik der Parallelprojektion wagen. Damit die Grafik flott gezeichnet werden kann, müssen wir die Berechnung der Ebenenpunkte vorher vornehmen und die Ergebnisse in Variablenfeldern speichern. Für jeden Eckpunkt des Würfels legen wir daher zwei (*x* und *y*) solcher Felder an. Die Berechnung selber könnte, durch die halbe Diagonale, einmal mit Sinus und einmal mit Cosinus, recht einfach gestaltet werden. Da der Würfel eine Sonderform darstellt, wollen wir uns nicht verzetteln und bleiben bei den beiden bekannten Formeln. Bisher haben wir noch nicht festgelegt, um welche Achse sich der Würfel drehen soll. Da das Koordinatenkreuz in der Mitte liegt, könnten wir theoretisch jede der drei Achsen nehmen. Es bietet sich aber die Y-Achse an, da wir dabei nur die Werte von 0 bis 89 Grad berechnen müssen und diese hintereinander ablaufen lassen können.

Wir drehen daher den Würfel um die Y-Achse und kippen ihn dabei um 10 Grad über die X-Achse. Nun kommen wir zum zeitkritischen Teil, zur Zeichnung der Grafik. Ich habe dazu einige Zeitmessungen vorgenommen, die ich Ihnen nicht vorenthalten will. Sie können die Erkenntnisse sicherlich in Ihren eigenen Programmen verwerten. Zum Löschen des Bildschirmes war die LINE-Anweisung mit *bf* in der Hintergrundfarbe am langsamsten, gefolgt von CLS. Am schnellsten wurde der Bildschirm mit *SetRast* in der Hintergrundfarbe freigemacht. Die folgenden Zeiten gelten für eine volle Drehung des Würfels um 90 Grad. Das Zeichnen der 90 Grafiken mit AREA im Outline-Modus dauerte ca. 5,9 und mit der LINE-Anweisung etwa 4,4 Sekunden. Etwas schneller waren die Library-Routinen *Move* und *Draw*. Damit konnte die Zeit auf 4,2 Sekunden gedrückt werden. Das ergibt zum Löschen und erneuten Zeichnen einer Grafik eine Zeit von nur 0,047 Sekunden. Schreiben wir zum Schluß alles in der richtigen Reihenfolge nieder:

```
REM WireCube  Pfad: Darstellung/9dreiD.Grafik/WireCube
'P9-2
DEFINT a-z
```

```
d=89
DIM x1(d),x2(d),x3(d),x4(d),x5(d),x6(d),x7(d),x8(d)
DIM y1(d),y2(d),y3(d),y4(d),y5(d),y6(d),y7(d),y8(d)
LIBRARY ":bue/graphics.library"
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
SCREEN 1,32Ø,sh,1,1: WINDOW 2,,,Ø,1
max=sh-6Ø

start:
  LOCATE 1,1:PRINT "Kantenlaenge (max."max"):"
  LOCATE 1,26:INPUT b
  IF b>max THEN start

LOCATE 14,1Ø:PRINT "bitte etwas Geduld"
b2=b/2  :rp&=WINDOW(8)
breite=WINDOW(2):hoehe=WINDOW(3)
xk=breite/2:yk=hoehe/2       'Nullpunkt  Koordinatenkreuz
pi!=3.141593:ww!=pi!/18Ø     'Umrechnung von Bogenmass

DEF FNxe(wi,xr,zr)=xk+zr*swy!+xr*cwy!
DEF FNye(wi,xr,yr,zr)=yk-(yr*cwx!+xr*swx!*swy!-zr*swx!*cwy!)

rechnen:
  swx!=SIN(1Ø*ww!)
  cwx!=COS(1Ø*ww!)
  FOR i = Ø TO 89
    swy!=SIN(i*ww!)
    cwy!=COS(i*ww!)
    x1(i)=FNxe(i,-b2,b2)    :y1(i)=FNye(i,-b2,-b2,b2)
    x2(i)=FNxe(i,-b2,-b2)   :y2(i)=FNye(i,-b2,-b2,-b2)
    x3(i)=FNxe(i,b2,-b2)    :y3(i)=FNye(i,b2,-b2,-b2)
    x4(i)=FNxe(i,b2,b2)     :y4(i)=FNye(i,b2,-b2,b2)
    x5(i)=x1(i)             :y5(i)=FNye(i,-b2,b2,b2)
    x6(i)=x2(i)             :y6(i)=FNye(i,-b2,b2,-b2)
    x7(i)=x3(i)             :y7(i)=FNye(i,b2,b2,-b2)
    x8(i)=x4(i)             :y8(i)=FNye(i,b2,b2,b2)
  NEXT i

Hauptprogramm:
  CLS: PALETTE Ø,Ø,Ø,Ø:PALETTE 1,1,Ø,Ø
  z=-1
  WHILE z
    FOR n=Ø TO 89 'STEP 5
```

```
      CALL SetRast&(rp&,Ø)
      CALL Move&(rp&,x5(n),y5(n))
      CALL Draw&(rp&,x1(n),y1(n))
      CALL Draw&(rp&,x2(n),y2(n))
      CALL Draw&(rp&,x3(n),y3(n))
      CALL Draw&(rp&,x4(n),y4(n))
      CALL Draw&(rp&,x1(n),y1(n))
      CALL Move&(rp&,x2(n),y2(n))
      CALL Draw&(rp&,x6(n),y6(n))
      CALL Draw&(rp&,x7(n),y7(n))
      CALL Draw&(rp&,x8(n),y8(n))
      CALL Draw&(rp&,x5(n),y5(n))
      CALL Draw&(rp&,x6(n),y6(n))
      CALL Move&(rp&,x3(n),y3(n))
      CALL Draw&(rp&,x7(n),y7(n))
      CALL Move&(rp&,x4(n),y4(n))
      CALL Draw&(rp&,x8(n),y8(n))
      FOR i=Ø TO 1ØØ:NEXT
    NEXT n
  ta-=INKEY-:IF ta-<>"" THEN z=Ø
  WEND
ende:
  WINDOW CLOSE 2:SCREEN CLOSE 1
  END
```

Das Ergebnis kann sich sehen lassen. Völlig ruckfrei und doch recht flott dreht sich der Würfel im Raum. Übrigens, wenn Sie die Drehung noch flotter haben wollen, dann entfernen Sie doch einfach das Hochkomma in der inneren FOR/NEXT-Schleife des Hauptprogrammes.

9.3.2 Ein tanzender Würfel mit farbigen Flächen

Das mit dem Drahtmodell funktioniert ja ganz gut. Doch wie sieht es mit einem richtigen Würfel mit farbigen Flächen aus? Schließlich finden wir in der Wirklichkeit nur selten solche Drahtmodelle vor. Auch diese Fragestellung soll uns nicht schrecken. Gleichzeitig wollen wir dabei die Grafik auf dem Bildschirm verschieben.

Überlegen wir dazu, was wir an unserem letzten Programm ändern müssen, um die neuen Forderungen zu erfüllen. Da der Würfel farbige Flächen bekommen soll, benötigen wir einen entsprechenden Screen mit der Bitmap-Tiefe 3. Bis zur Berechnung der Eckpunkte des Würfels im Raum bleibt ansonsten alles gleich. Weil wir die Grafik verschieben wollen, müssen wir die Koordinatenwerte *kx* und *ky* aus den Variablen-Feldern der Ebenenkoordinaten herausnehmen.

Das Hauptproblem liegt darin, ein schnelles Füllen der Flächen zu verwirklichen. Am einfachsten und schnellsten läßt sich das mit den AREA-Anweisungen realisieren. Leider ergibt das kein vernünftiges Bild. Durch das unentwegte Löschen und Füllen der Grafik werden die Flächen flackernd und mehrfarbig gestreift ausgegeben. Da hilft nur eines: DoubleBuffering. Wir zeichnen die Grafik in einem verdeckten Window und kopieren diese, sobald sie fertiggestellt ist, in das sichtbare Fenster. Den Kopiervorgang übernimmt wieder die blitzschnelle Kopier-Routine *ClipBlit*. Dazu benötigen wir die RastPort-Adressen der beiden Fenster. Wir finden sie in der 50. Speicherstelle der Window-Strukturen.

Bleibt noch die Frage offen, wie die Flächen des Würfels die richtigen Farben bekommen. Nur mit der korrekten Reihenfolge der Farben kann die Drehung vom Anwender auch optisch akzeptiert werden. Die obere Fläche ist kein Problem. Da sie immer zu sehen ist, füllen wir sie mit einer eigenen AREA-Folge mit der Farbnummer 1. Die untere Fläche vergessen wir, da sie nicht zu sehen ist. Von den restlichen 4 Flächen sind immer 2, in unterschiedlicher Größe, sichtbar. Da wir wieder 4 einzelne 90-Grad-Drehungen hintereinander zeichnen, stellen wir genau genommen auch immer nur die beiden Flächen dar. Damit haben wir auch bereits die Lösung für unser Problem. Wir färben die beiden sichtbaren Flächen immer um eine Farbnummer versetzt ein. Bei der vierten Farbe würde es zu einer Überschneidung kommen. Um eine weitere Abfrage in der Schleife zu vermeiden, setzten wir daher die Farbe 6 auf die gleichen Farbwerte wie die Farbe 2.

Damit haben wir die wichtigsten Probleme bereits gelöst. Bleibt nur noch die Bewegung des Würfels durch Erhöhung der Werte des Koordinatenkreuzes. Mit zwei IF-Anweisungen können wir die Fenstergrenzen abfragen und zur Verhinderung einer Kollision die Variablen für die Bewegung *dx* oder *dy* mit –1 multiplizieren. Damit wird die Richtung um 180 Grad verändert. Mit Kollisionsabfrage ist in diesem Fall gemeint, daß ein Überschreiten der Fenstergrenzen unbedingt verhindert werden muß. Die AREA-Anweisung mit ungültigen Werten führt zum Programmabbruch.

```
REM DancingCube  Pfad: Darstellung/9dreiD.Grafik/DancingCube
'P9-3
IF FRE(-1)<9ØØØØ& THEN PRINT "Speicher zu klein":END
DEFINT a-z
d=89
DIM x1(d),x2(d),x3(d),x4(d),x5(d),x6(d),x7(d),x8(d)
DIM y1(d),y2(d),y3(d),y4(d),y5(d),y6(d),y7(d),y8(d)
LIBRARY ":bue/graphics.library"
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
SCREEN 1,32Ø,sh,3,1: WINDOW 2,,,16,1
rp2&=PEEKL(WINDOW(7)+5Ø):max=sh-1ØØ
```

```
WINDOW 3,,,Ø,1
rp3&=PEEKL(WINDOW(7)+5Ø)
COLOR 5:LOCATE 1Ø,12:PRINT "dancing   CUBE":COLOR 1

start:
  LOCATE 1,1:PRINT "Kantenlaenge (max."max"):"
  LOCATE 1,26:INPUT b
  IF b>max THEN start

LOCATE 18,1Ø:PRINT "bitte etwas Geduld"
b2=b/2
di=SQR(b^2+b^2)/2+5          'halbe Diagonale
breite=WINDOW(2):hoehe=WINDOW(3)
xk=breite/2:yk=hoehe/2       'Nullpunkt  Koordinatenkreuz
pi!=3.141593:ww!=pi!/18Ø     'Umrechnung von Bogenmass
DEF FNxe(wi,xr,zr)=zr*swy!+xr*cwy!
DEF FNye(wi,xr,yr,zr)=yr*cwx!+xr*swx!*swy!-zr*swx!*cwy!

rechnen:
  swx!=SIN(15*ww!)
  cwx!=COS(15*ww!)
  FOR i = Ø TO 89
    swy!=SIN(i*ww!)
    cwy!=COS(i*ww!)
    x1(i)=FNxe(i,-b2,b2)     :y1(i)=FNye(i,-b2,-b2,b2)
    x2(i)=FNxe(i,-b2,-b2)    :y2(i)=FNye(i,-b2,-b2,-b2)
    x3(i)=FNxe(i,b2,-b2)     :y3(i)=FNye(i,b2,-b2,-b2)
    x4(i)=FNxe(i,b2,b2)      :y4(i)=FNye(i,b2,-b2,b2)
    x5(i)=x1(i)              :y5(i)=FNye(i,-b2,b2,b2)
    x6(i)=x2(i)              :y6(i)=FNye(i,-b2,b2,-b2)
    x7(i)=x3(i)              :y7(i)=FNye(i,b2,b2,-b2)
    x8(i)=x4(i)              :y8(i)=FNye(i,b2,b2,b2)
  NEXT i

Hauptprogramm:
  CLS: PALETTE Ø,Ø,Ø,Ø:PALETTE 2,Ø,1,Ø :PALETTE 6,Ø,1,Ø
  z=-1 :fa=1:dx=3:dy=3
  WHILE z
    fa=fa+1:IF fa>5 THEN fa=2
    WINDOW OUTPUT 2
    FOR n=Ø TO 89 'STEP 5
      IF xk-di<=Ø OR xk+di>=breite THEN dx=dx*(-1)
      IF yk-di<=Ø OR yk+di>=hoehe  THEN dy=dy*(-1)
```

```
      xk=xk+dx:yk=yk+dy
      CLS
      COLOR 1
        AREA(xk+x5(n),yk-y5(n)):AREA(xk+x6(n),yk-y6(n))
        AREA(xk+x7(n),yk-y7(n)):AREA(xk+x8(n),yk-y8(n))
        AREAFILL Ø
      COLOR fa
        AREA(xk+x1(n),yk-y1(n)):AREA(xk+x4(n),yk-y4(n))
        AREA(xk+x8(n),yk-y8(n)):AREA(xk+x5(n),yk-y5(n))
        AREAFILL Ø
      COLOR fa+1
        AREA(xk+x2(n),yk-y2(n)):AREA(xk+x1(n),yk-y1(n))
        AREA(xk+x5(n),yk-y5(n)):AREA(xk+x6(n),yk-y6(n))
        AREAFILL Ø
      CALL clipBlit&(rp2&,Ø,Ø,rp3&,Ø,Ø,breite,hoehe,192)
    NEXT n
    WINDOW OUTPUT 3
    ta$=INKEY$:IF ta$<>"" THEN z=Ø
  WEND

ende:
  WINDOW CLOSE 3:WINDOW CLOSE 2:SCREEN CLOSE 1
  END
```

Wie gefällt Ihnen der tanzende Würfel. Sicherlich genausogut wie mir. Sie können auch in diesem Programm die Drehung des Würfels um den Faktor 5 beschleunigen, indem Sie das Hochkomma in der FOR/NEXT-Schleife des Hauptprogrammes entfernen.

9.4 Funktionen in drei Dimensionen

Gleichungen mit zwei Unbekannten sind Ihnen sicherlich ein Begriff. Einige einfache Formen wie y = cos(x), y = sin(x) und y = tan(x) haben wir bereits am Anfang des Buches grafisch in zwei Dimensionen dargestellt. Die allgemeine Schreibweise einer solchen Funktion ist y = f(x). Der Wert y ist also von einem anderen Wert x abhängig.

Wenn allerdings der Wert y von zwei verschiedenen Werten x und z abhängig ist, geht mit der zweidimensionalen Darstellung nichts mehr. Man nennt das eine Gleichung mit drei Unbekannten. Die Funktion hat dann die allgemeine Form y = f(x,y). Ein sehr einfaches Beispiel ist die Berechnung einer Seite eines rechtwinkeligen Dreieckes aus den beiden anderen Seiten. Nach dem Satz des Pytagoras lautet die Formel dazu y = SQR(x^2 + y^2). Um diese Formel grafisch darstellen zu können, benötigen wir drei

Achsen, also eine dreidimensionale Grafik. In diese Grafik tragen wir verschiedene Werte, innerhalb bestimmter Grenzen, für die Parameter *X* und *Z* ein. Dadurch ergeben sich verschiedene Kurven, die übereinandergelagert gezeichnet werden. Die einzelnen Werte kann man entweder einfach als Punkte zeichnen oder die einzelnen Punkte durch Linien zu einem Netz verbinden. Die Felder dazwischen kann man farbig füllen und verdeckte Punkte löschen.

Die grafische Darstellung solcher dreidimensionaler Gleichungen ist aus der Mathematik nicht mehr wegzudenken. So kann man bei komplexen Gleichungen aus dem Verlauf der Grafik Rückschlüsse ziehen und notwendige Berechnungen gezielt auf bestimmte Bereiche konzentrieren. Uns interessiert allerdings weniger der mathematische Aspekt, sondern die programmtechnischen Erfordernisse. Drehen wir dieses Mal den Spieß um und schauen uns anhand eines Programmes an, wie man die einzelnen Probleme lösen kann.

```
REM 3DFunktion  Pfad: Darstellung/9dreiD.Grafik/3DFunktionen
'P9-4
'Hier koennen andere Parameter eingesetzt werden
'################
DEF FNFunktion(x,z) =SQR(z^2+x^2)  'Rechenfunktion
xb% = 60        'Anzahl der Berechnungen fuer X
zb% = 15        'Anzahl der Berechnungen fuer Z
'################
DIM xe(zb%,xb%),ye(zb%,xb%)
pi=3.141593

'eingeben
  INPUT "maximalen X-Wert: ";xmax
  INPUT "minimalen X-Wert: ";xmin
  INPUT "maximalen Z-Wert: ";zmax
  INPUT "minimalen Z-Wert: ";zmin

'Abstufung berechnen
  stufX=(xmax-xmin)/(xb%-1)       'Abstufung der X-Werte
  stufZ=(zmax-zmin)/(zb%-1)       'Abstufung der Z-Werte

'ScreenEinrichten
  sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
  SCREEN 1,320,sh,4,1
  WINDOW 2,,,0,1
  PALETTE 0,.6,.6,.6
  FOR i=1 TO 15
    PALETTE i,i*1/15,0,0
  NEXT
  PRINT "ich rechne"
```

```
breite%=WINDOW(2)-3
hoehe%=WINDOW(3)-3
xw=2Ø                   'Drehwinkel X-Achse
yw=2Ø                   'Drehwinkel Y-Achse
xws=SIN(xw*pi/18Ø)      'Sinus Drehwinkel X-Achse
yws=SIN(yw*pi/18Ø)      'Sinus Drehwinkel Y-Achse
xwc=COS(xw*pi/18Ø)      'Cosinus Drehwinkel X-Achse
ywc=COS(yw*pi/18Ø)      'Cosinus Drehwinkel Y-Achse

'Raumkoordinaten berechnen
'Ebenenkoordinaten berechnen
  FOR i=Ø TO zb%-1
    zr=zmin+stufZ*i:zr=-zr
    FOR j=Ø TO xb%-1
      xr=xmin+stufX*j
      yr=FNFunktion(xr,zr)
      xe(i,j)=zr*yws+xr*ywc
      ye(i,j)=yr*xwc+xr*xws*yws-zr*xws*ywc
      IF xeMax<xe(i,j) THEN xeMax=xe(i,j)
      IF xeMin>xe(i,j) THEN xeMin=xe(i,j)
      IF yeMax<ye(i,j) THEN yeMax=ye(i,j)
      IF yeMin>ye(i,j) THEN yeMin=ye(i,j)
    NEXT
  NEXT

'Skalierung berechnen/Bildschirmgroesse anpassen
  xFakt=breite%/(xeMax-xeMin)
  yFakt=hoehe%/(yeMax-yeMin)

'Lage des Koordinaten-Nullpunktes berechnen
  xk=Ø-(xeMin*xFakt)
  yk=hoehe%+(yeMin*yFakt)

'Zeichnen
  CLS :fa=1
  FOR i=Ø TO zb%-1
    COLOR fa
    FOR j=Ø TO xb%-1
      PSET ((xk+(xe(i,j))*xFakt),(yk-(ye(i,j))*yFakt))
    NEXT
    fa=fa+1:IF fa>15 THEN fa=1
  NEXT

GOSUB Zeichnen2
```

```
WHILE INKEY$="":WEND

WINDOW CLOSE 2
SCREEN CLOSE 1
END

Zeichnen2:
  ti&=TIMER:WHILE ti&+3>TIMER:WEND
  CLS :fa=1
  FOR i=Ø TO zb%-1
    COLOR fa
    FOR j=1 TO xb%-1
      LINE ((xk+(xe(i,j-1))*xFakt),(yk-(ye(i,j-1))*yFakt))
**            -((xk+(xe(i,j))*xFakt),(yk-(ye(i,j))*yFakt))
    NEXT
    fa=fa+1:IF fa>15 THEN fa=1
  NEXT
  COLOR 1
  FOR j=Ø TO xb%-1
    FOR i=1 TO zb%-1
      LINE ((xk+(xe(i-1,j))*xFakt),(yk-(ye(i-1,j))*yFakt))
**            -((xk+(xe(i,j))*xFakt),(yk-(ye(i,j))*yFakt))
    NEXT
  NEXT
RETURN
```

Da es sich um ein Programm zur dreidimensionalen Darstellung von Funktionen handelt, fangen wir gleich mit der Rechenformel an. Das Programm ist natürlich so angelegt, daß an dieser Stelle eine beliebige Funktion eingetragen werden kann. Auch die Anzahl der einzelnen Berechnungen für X und Z können hier geändert werden. Um durch eine zu hohe Dimensionierung ein *Out Of Memory* des Basic-Interpreters zu vermeiden, werden diese Werte nicht durch INPUT vom Anwender geholt. Die maximalen und minimalen Werte für X und Z werden anschließend vom Anwender erfragt. Aus den eingegebenen Werten und der Anzahl der Berechnungen können wir bereits die Abstufung bzw. die Rechenschritte für X und Z errechnen.

Etwas später als gewohnt richten wir dann einen Bildschirm der Tiefe 4 ein, öffnen ein Fenster mit den maximal möglichen Abmessungen und stufen die Farben 1 bis 15 mit einem steigenden Rotanteil ab. Gleich anschließend legen wir die Drehwinkel für die X- und Y-Achse mit 20 Grad fest und berechnen die Sinus- und Cosinus-Werte. Bei den Drehwinkeln können Sie etwas experimentieren, bis die Grafik so gezeichnet wird, wie Sie es sich vorstellen.

Die Raumkoordinaten und die Ebenenkoordinaten der einzelnen Punkte berechnen wir in den gleichen Schleifen. Für die Raumkoordinaten *zr* und *xr* setzen wir die Abstufungen ein, die wir zu Programmbeginn berechnet haben. Mit den Werten der Raumkoordinaten *zr* und *xr* rufen wir die Funktion auf und erhalten *yr* zurück. Zur Berechnung der Ebenenkoordinaten stützen wir uns wieder auf die beiden bekannten Formeln. Allerdings lassen wir dabei die Variablen *xk* und *yk* für den Koordinaten-Nullpunkt außer acht. Dabei zeichnen wir die Punkte noch nicht, sondern speichern sie in den Feld-Variablen *xe()* und *ye()* ab. Warum das so ist, werden Sie gleich erfahren. Gleichzeitig berechnen wir in den beiden ineinandergeschachtelten FOR/NEXT-Schleifen die maximalen und die minimalen Werte der Ebenenkoordinaten.

Sie haben etwas weiter oben gelesen, daß der Anwender beliebige Werte als Obergrenzen für X und Z eingeben kann. Da können wir natürlich schlecht verlangen, wie das leider in vielen Programmen dieser Art üblich ist, daß er schnell im Kopf nachrechnet, wie die Grafik auf den Bildschirm paßt. Dazu berechnen wir je einen Multiplikator, der die Grafik genau der Bildschirmgröße anpaßt. Jetzt wissen Sie auch, warum wir in der Hauptrechnung die maximalen und minimalen Werte der Ebenenkoordinaten berechnet haben. Mit der richtigen Größe der Grafik ist es jedoch noch nicht getan. Bis jetzt liegt die Grafik wahrscheinlich in einem Bereich irgendwo außerhalb der Bildschirmgrenzen. Richtig, uns fehlt noch die Lage des Koordinaten-Nullpunktes.

Nachdem wir auch noch den Nullpunkt berechnet haben, steht nichts mehr im Wege die Grafik zu zeichnen. Dank der gespeicherten Werte geht das recht flott vonstatten. Die Farbänderung vom dunklen zum hellen Rot soll den optischen Eindruck der dritten Dimension verstärken.

Recht eindrucksvoll ist eine andere Darstellungsform der Grafik. Dabei werden die einzelnen Punkte der Grafik durch Linien verbunden. Man nennt diese Vernetzung der Grafik auch *Crosshatching*. Programmtechnisch gesehen ist die Zeichnung des Netzes kein Problem. In unserem Programm wird deshalb anschließend an die Punktgrafik, nach einer kleinen Warteschleife, die Netzgrafik gezeichnet.

Kapitel 10

Animation

Unter Animation bei der Computer-Grafik versteht man heute jede Art von Bewegung auf dem Bildschirm. Unter diesem Gesichtspunkt haben Sie in den zurückliegenden Kapiteln schon reichlich über Animation erfahren. Als die Computertechnik noch in den Kinderschuhen steckte, also die Zeit vor dem Amiga, kannte man noch keine Befehle, mit denen man blitzschnell ganze Bitmaps kopieren konnte. Man schaffte hardwareseitig die Voraussetzungen, um wenigstens kleine Bilder über den Bildschirm schieben zu können. Diese Objekte nennt man *Sprites*. Natürlich kann der Amiga ebenfalls solche Objekte erzeugen. Auch wenn es andere Möglichkeiten der Objektverschiebung gibt, auf die bewegten Objekte kann man nicht verzichten. Das liegt hauptsächlich daran, daß sie beim Amiga vollkommen unabhängig vom normalen Programm gesteuert werden können. Der Amiga stellt zwei Arten dieser Objekte zur Verfügung, BOBs und Sprites. Die Sprites werden von der Hardware kontrolliert, die BOBs dagegen im wesentlichen von der Software. Diese einfache Unterscheidung genügt für die Programmierung der Objekte mit den normalen Befehlen des Amiga-Basic. Wollen wir aber mehr über die beweglichen Objekte erfahren, so greifen wir auf die Routinen der Libraries zurück. Unter diesem Gesichtspunkt reicht die genannte Unterscheidung der Objekte nach BOBs und Sprites nicht aus. Aus den beiden Objekt-Arten werden plötzlich fünf. Im einzelnen werden vom System folgende Animations-Objekte unterschieden:

`VSprites`
sind virtuelle (dem Wesen nach) Sprites.

`BOBs`
sind grafische Objekte, die vor allem von der Software gesteuert werden.

`Simple Sprites`
sind Hardware-Sprites.

`Mauszeiger`
ist der Hardware-Sprite Null.

`AnimObjects`
sind zusammengehörende BOBs für einen kontinuierlichen Bewegungsablauf.

10.1 VSprites

Stören Sie sich bitte nicht an dem V vor dem Wort Sprites. Das V sagt Ihnen, daß die Sprites des Amiga-Basic keine Hardware-Sprites sind. Das wäre auch nicht besonders erfreulich. Sie könnten nur 8 Sprites darstellen und müßten auch auf viele Vorteile verzichten, die die Programmierung der Sprites so einfach und vielseitig gestalten. Das V bedeutet virtueller Sprite. Ihm wird nur temporär ein Hardware-Sprite zugewiesen. Ein VSprite vereinigt daher die Vorteile eines Hardware-Sprites mit denen eines softwaremäßig unterstützten Objektes. So lassen sich zum Beispiel die Sprites durch die Hardware mit einer schier unglaublichen Geschwindigkeit über den Bildschirm bewegen. Gleichzeitig wird durch die Software jede Berührung mit einem anderen Objekt erkannt.

Auf einer horizontalen Ebene des Bildschirmes lassen sich maximal 8 Sprites darstellen. Die Breite eines Sprites ist auf Wortlänge, also 16 Grafikpunkte begrenzt. Diese 16 Pixel gelten aber nur für einen Bildschirm mit niedriger Auflösung. Bei einem Hires-Screen werden die Sprites doppelt so breit dargestellt. Natürlich ändert sich dabei die Auflösung nicht. Die Höhe können Sie im Rahmen der Bildschirmhöhe beliebig bestimmen. Jeder Sprite hat 4 Farben zur Verfügung, die jeweils unterste Farbe ist transparent. Die Sprite-Farben werden in den Objekt-Daten mitgegeben und sind unabgängig von den Farben des Screens.

Die Grafik der Sprites wird in einer Zeichenkette gespeichert. Zum Zeichnen der Sprites und BOBs verwenden Sie am einfachsten einen Object-Editor (z.B. den von Ihrer Original-Diskette mit dem Amiga-Basic), der Ihre Sprites in einer Datei ablegt. Auf der diesem Buch beiliegenden Diskette, in der Schublade *Bild*, befindet sich ein einfaches Sprite (besser gesagt die Datei eines Sprites) mit dem Namen *Sprite1*, der für Ihre ersten Versuche genügen dürfte.

Kommen wir nun zu den einzelnen Basic-Befehlen für die Objekte. Der Begriff Objekte ist bewußt gewählt, denn alle folgenden Befehle gelten sowohl für Sprites als auch für BOBs. Mit der Anweisung

```
OBJECT.SHAPE Objektnummer,String
```

wird ein neues Objekt erstellt und mit einer Nummer zur Identifizierung versehen. Zuvor muß natürlich die Zeichenkette mit den Daten vorhanden sein.

```
Beispiel: OPEN ":Bild/Sprite1" FOR INPUT AS 1
          OBJECT.SHAPE 1,INPUT$(LOF(1),1)
```

In der zweiten Schreibweise

`OBJECT.SHAPE ObjektnummerNeu,ObjektnummerVorhanden`

wird ein neues Sprite *ObjektnummerNeu* geschaffen, das die Daten aus dem Sprite *ObjektnummerVorhanden* übernimmt.

`Beispiel: FOR i=2 TO n :OBJECT.SHAPE i,1 :NEXT`

Um dem Objekt auf den Bildschirm seine Position zuzuweisen, wird das Befehlspaar

```
OBJECT.X Objektnummer,X
OBJECT.Y Objektnummer,Y
```

eingesetzt. Die Koordinaten werden wie üblich von der oberen linken Ecke des Bildschirmes bis zur oberen linken Ecke des Objektes in Bildpunkten gemessen.

`Beispiel: FOR i=1 TO n:OBJECT.X i,Ø:OBJECT.Y i,2Ø*i+2Ø: NEXT`

Um das Objekt sichtbar zu machen, wird

`OBJECT.ON Objektnummer, Objektnummer, ...`

angewendet. Ebenso kann es mit

`OBJECT.OFF Objektnummer, Objektnummer, ...`

wieder vom Bildschirm verbannt werden. Die Anweisungen gelten mit dem Parameter *Objektnummer* für einzelne, oder wenn kein Parameter angegeben ist, für alle Objekte auf dem Bildschirm.

Für die Geschwindigkeit der Objekte setzen Sie die beiden Befehle

```
OBJECT.VX Objektnummer, Geschwindigkeit
OBJECT.VY Objektnummer, Geschwindigkeit
```

ein. Die Geschwindigkeit wird in Pixel pro Sekunde gemessen. Positive Werte bewegen das Objekt nach rechts (VX) oder nach unten (VY), und negative Werte rufen eine Bewegung in entgegengesetzter Richtung hervor.

`Beispiel: FOR i= 1 TO n:OBJECT.VX i,1:NEXT`

Es genügt aber nicht, nur die Geschwindigkeit der Objekte festzulegen. Damit Bewegung in die Objekte kommt, muß noch ein

`OBJECT.START Objektnummer, Objektnummer, ...`

programmiert werden. Sich bewegende Objekte werden mit

`OBJECT.STOP Objektnummer, Objektnummer, ...`

wieder angehalten. Beide Anweisungen gelten mit dem Parameter *Objektnummer* wieder für einzelne Objekte, oder ohne diese Angabe für alle aktiven Objekte.

Besonders interessant ist, daß den Objekten auch eine Beschleunigung mitgegeben werden kann. Das Anweisungspaar

```
OBJECT.AX Objektnummer, Beschleunigung
OBJECT.AY Objektnummer, Beschleunigung
```

legt für ein Objekt die Beschleunigung in Pixel pro Sekunde in X- und/oder Y-Richtung fest.

```
Beispiel: OBJECT.AX 2,beschl
```

Für die Programmierung kann es oft nützlich sein, die aktuellen Positionen der Objekte in Erfahrung zu bringen. Auch das ist für das Amiga-Basic kein Problem. Mit den Funktionen

```
OBJECT.X(Objektnummer)
OBJECT.Y(Objektnummer)
```

erhalten Sie die aktuellen Koordinaten des Objektes *Objektnummer.*

Sogar die Geschwindigkeit (in Bildpunkten pro Sekunde) von einzelnen Objekten können Sie abfragen. Die Funktionen

```
OBJECT.VX(Objektnummer)
OBJECT.VY(Objektnummer)
```

liefern die aktuelle Geschwindigkeit in X- oder Y-Richtung des Objektes mit der Kennung *Objektnummer.*

```
Beispiel: IF OBJECT.X(i)>=53Ø THEN g=OBJECT.VX(i)
```

Nun brauchen wir nur noch die einzelnen Anweisungen nacheinander aufzuschreiben, und schon haben wir ein kleines, aber hübsches Animations-Programm.

```
REM Truck Race  Pfad: Darstellung/1ØAnimation/TruckRace
'P1Ø-1
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
IF sh >2ØØ THEN n=2Ø ELSE n=16
SCREEN 1,64Ø,sh,2,2
WINDOW 2,,,1,1
OPEN ":Bild/Sprite1" FOR INPUT AS 1
OBJECT.SHAPE 1,INPUT-(LOF(1),1)
CLOSE 1 :PALETTE Ø,.6,.6,.6
FOR i=2 TO n :OBJECT.SHAPE i,1 :NEXT
start:  CLS
FOR i=1 TO n:OBJECT.X i,Ø:OBJECT.Y i,12*i-1Ø: NEXT
OBJECT.ON
```

```
FOR i= 1 TO n:OBJECT.VX i,1:NEXT
OBJECT.START
SOUND 2ØØØ,3,255:a=-1 :i=Ø
WHILE a
  i=i+1:IF i>n THEN i=1
  RANDOMIZE TIMER:beschl=RND*6
  OBJECT.AX i,beschl
  IF OBJECT.X(i)>53Ø THEN g=OBJECT.VX(i):OBJECT.STOP :a=Ø
WEND
PRINT "Sieger Truck Nummer: "i:PRINT "Ziel-Geschwindigkeit:"g
PRINT :PRINT  "Noch ein Rennen? j/n"
taste: ta-=INKEY-:IF ta-="" THEN taste
IF ta-="j" THEN start ELSE OBJECT.OFF
WINDOW CLOSE 2:SCREEN CLOSE 1
```

Die einzelnen Befehle können Sie leicht nach den vorangegangenen Erklärungen nachvollziehen. Der Clou des Programmes liegt in der WHILE/WEND-Schleife. Um wirkliche Zufallszahlen zu erhalten, wird der Timer mit RANDOMIZE TIMER bei jedem Schleifendurchgang neu gestartet. Die Beschleunigung der einzelnen Lastwagen erhält dadurch einen zufälligen Wert zwischen 0 und 6. Da die Beschleunigungswerte laufend neu gesetzt werden, erhalten wir einen echten Rennverlauf.

Wie gefällt Ihnen das erste Animations-Programm? Sie können ja vor dem Start auf den Sieger Wetten abschließen. Das Ergebnis ist jedenfalls rein zufällig.

10.2 BOBs

Mit den BOBs stehen Ihnen weitere Objekte zur Verfügung, die Sie separat oder zusätzlich zu den Sprites auf den Bildschirm bringen können. (Sie kennen vielleicht die Bezeichnung Shapes von anderen Computern, die mit den BOBs vergleichbar sind.) Die Bezeichnung BOB ist eine Abkürzung von *Blitter OBject*. Damit ist schon angedeutet, wer die Verantwortung für die BOBs trägt. Der Spezial-Chip Blitter kann Grafikbereiche unheimlich schnell kopieren. Zusammen mit der Software des Amiga werden damit die BOBs über den Bildschirm bewegt. Allerdings liegt die Geschwindigkeit, mit der die BOBs bewegt werden können, deutlich niedriger als bei den Sprites.

Im Gegensatz zu den Sprites ist die Anzahl gleichzeitig darstellbarer BOBs nicht beschränkt. Auch die Größe findet nur eine Einschränkung in der Bildschirmgröße. Die Anzahl der Farben richtet sich nach dem Screen, auf dem sie dargestellt werden. Auf einem Bildschirm mit einer Bitmap-Tiefe von 5 können also 32farbige BOBs dargestellt werden. BOBs mit einer geringeren Farbtiefe als der Screen werden in den ersten Farben der Farbtafel dargestellt. Bei einer größeren Farbtiefe der BOBs gegenüber der Screen-

Bitmap werden diese zwar dargestellt, aber nur in der Anzahl der Farben, die der Screen zur Verfügung stellt. Die Zusammensetzung der Farben kann man ändern. Wir kommen darauf noch zu sprechen.

Zum Testen der einzelnen Anweisungen und Funktionen stehen Ihnen drei einfache BOBs mit den Namen BOB1, BOB2 und BOB3, in der Schublade *Bild* der Buch-Diskette, zur Verfügung. – Bei den in diesem Kapitel folgenden Befehlsbeschreibungen sagt auch hier die Bezeichnung *Objekt*, daß der Befehl gleichermaßen für BOBs und Sprites gilt.

10.2.1 Objekt-Kollision

Kommen wir nun zum interessantesten Aspekt der Objekt-Programmierung, zu der Begegnung und der Kollision von Objekten. Die System-Software des Amiga kann erkennen, wenn sich Objekte auf dem Bildschirm begegnen, oder wenn sie die Fenstergrenzen berühren. Erst damit haben Sie die Voraussetzung, um mit den Objekten eine schnelle und einfache Programmierung von Spielen, Grafiksimulationen u.ä. zu realisieren. Wie wollen Sie sonst in einem Schießspiel feststellen, ob der Laserschuß ein anderes Objekt getroffen hat? Wie könnte sonst in einer Drehbank-Simulation die Berührung von Drehmeißel und Werkstück festgestellt werden? Dazu müßte sonst ein Riesenaufwand an Abfragen programmiert werden, die zusätzlich viel zu viel Zeit verschlingen würden.

Beim Amiga werden die oben genannten Kollisionen in einer Warteschlange abgelegt. Diese gilt es nun abzufragen, um programmgesteuert darauf reagieren zu können. Gehen wir wieder der Reihe nach vor. Zu den Unterbrechungsereignissen, wie TIMER MOUSE u.ä., gehört auch die Anweisung

```
ON COLLISION GOSUB Sprungmarke
```

Mit diesem Befehl legen Sie fest, daß bei jeder Kollision zu der Sprungmarke verzweigt werden soll. Dort kann dann näheres über die Kollision abgefragt und darauf reagiert werden. Wie bei diesen Anweisungen üblich, müssen diese mit einem eigenen Befehl

```
COLLISION ON
```

aktiviert werden. Wollen Sie vorübergehend keine Kollisionen gemeldet haben, so bringt

```
COLLISION STOP
```

die gewünschte Wirkung. Mit

```
COLLISION OFF
```

schalten Sie die Unterbrechungsreaktionsfähigkeit wieder aus. Die letzte Anweisung sollten Sie am Programmende nicht vergessen. Andernfalls kann es passieren, daß sich das System aufhängt.

Wie bereits geschildert, wird normalerweise jede Kollision eines Objektes mit dem Fensterrand oder mit einem anderen Objekt gemeldet. Das Basic des Amiga stellt aber eine Anweisung zur Verfügung, mit der man die Kollisions-Meldungen nach Belieben vorher festlegen kann.

```
OBJECT.HIT Objektnummer,Selbstmaske,Fremdmaske
```

Wenn Sie den Begriff Maske hören, wissen Sie gleich, daß Sie sich dazu auf die Ebene der Bits begeben müssen. In der Selbstmaske ist das noch ganz einfach. Sie legen darin einfach nur fest, welches Bit für das Objekt zur Erkennung dienen soll. Die Maske wird in Wortlänge angegeben. Sie ist also 16 Bit breit, und damit kann die Kennung für 16 Objekte aufgenommen werden. In der Fremdmaske belegen Sie die Bits, die die Kennung der Objekte tragen, bei dessen Berührung Sie eine Kollisions-Meldung erhalten wollen. Für die Fenstergrenzen wird das Bit 0 gesetzt. Das System führt bei einer Kollision eine AND-Verknüpfung der Bits durch. Ergibt die Verknüpfung den Wert 1 (Bit gesetzt), so folgt eine Kollisions-Meldung, andernfalls unterbleibt sie. Damit Sie dazu ein praktisches Beispiel sehen können, untersuchen wir die Masken aus dem folgenden Programm *Rubber Ball.*

BOB	Selbstmaske	Wert	Fremdmaske	Wert	Kollision mit
1	0000000000000010	2	0000000001111001	121	Fenst,2,5,3,6
4	0000000000000100	4	0000000001111001	121	Fenst,2,5,3,6
2	0000000000001000	8	0000000001100111	103	Fenst,1,4,3,6
5	0000000000010000	16	0000000001100111	103	Fenst,1,4,3,6
3	0000000000100000	32	0000000000011111	31	Fenst,1,4,2,5
6	0000000001000000	64	0000000000011111	31	Fenst,1,4,2,5

Die Werte ergeben sich wie üblich aus der Summe der Bits als Potenz von 2. Berechnen wir als Beispiel BOB 1:

Selbstmaske: 2^1=2
Fremdmaske: 2^0+2^3+2^4+2^5+2^6=121

Die Anweisung lautet also: OBJECT.HIT 1,2,121.

Bei dem Beispielprogramm sind drei verschiedenfarbige Bälle in doppelter Anzahl vorhanden. Die Masken der 6 OBJECT.HIT-Anweisungen sind so aufgebaut, daß jede Kollision registriert wird, nur nicht der Zusammenstoß mit einem gleichfarbigen Ball. Nachdem wir nun wissen, wie eine Kollisions-Meldung ausgelöst wird, brauchen wir nur noch zu fragen, welches Objekt mit was zusammengestoßen ist. Dazu bietet das Amiga-Basic die Funktion

```
v=COLLISION(n)
```

Im Parameter *n* liegt der Schlüssel zur Lösung. Drei verschiedene Aufruf-Möglichkeiten kennt die Funktion:

Funktionsaufruf	Ergebnis der Funktion
COLLISION(0)	Nummer des Objektes, welches zuletzt kollidierte; keine Veränderung der Kollisions-Warteschlange.
COLLISION(-1)	Nummer des Fensters, in dem die durch COLLISION(0) festgestellte Kollision stattgefunden hat.
COLLISION(n>0)	Nummer des Objektes, das mit dem Objekt *n* zusammengestoßen ist. Die Information wird aus der Warteschlange gelöscht. Im einzelnen sind folgende Resultate möglich: –1 obere Fenstergrenze –2 linke Fenstergrenze –3 untere Fenstergrenze –4 rechte Fenstergrenze >0 Objektnummer

Wenn Sie von einer Kollision mit einer Fenstergrenze lesen, denken Sie natürlich an die sichtbaren Grenzen des Fensters. Es gibt aber auch ein Pseudo-Fenster, für das die gleichen Gesetzmäßigkeiten gelten. Mit der Anweisung

```
OBJECT.CLIP (x1,y1)-(x2,y2)
```

definieren Sie einen rechteckigen Bereich, den die Objekte nicht verlassen können, bzw. dessen Grenzen eine Kollisions-Meldung auslösen. Schauen Sie sich nun an einem praktischen Beispiel an, wie gut oder wie schlecht die Kollisions-Meldungen und die Kollisions-Abfragen funktionieren.

```
REM RubberBall  Pfad: Darstellung/1ØAnimation/RubberBall
'P1Ø-2
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):sw=64Ø:tiefe=3
SCREEN 1,sw,sh,tiefe,2
WINDOW 2,,,Ø,1
w=WINDOW(2) :w2=w/2:w4=w/4
h=WINDOW(3) :h2=h/2:h4=h/4
ON COLLISION GOSUB getroffen
PALETTE Ø,Ø,.6,Ø
FOR i=1 TO 3
```

```
  OPEN ":Bild/BOB"+RIGHT-(STR-(i),1) FOR INPUT AS 1
  OBJECT.SHAPE i,INPUT-(LOF(1),1)
  CLOSE 1
NEXT
FOR i=1 TO 3:OBJECT.SHAPE i+3,i:NEXT
OBJECT.PRIORITY 1,6
OBJECT.PRIORITY 4,7
'gleichfarbige Objekte sollen nicht kollidieren
  OBJECT.HIT 1,2,121
  OBJECT.HIT 4,4,121
  OBJECT.HIT 2,8,1Ø3
  OBJECT.HIT 5,16,1Ø3
  OBJECT.HIT 3,32,31
  OBJECT.HIT 6,64,31
OBJECT.CLIP (Ø,Ø)-(w,h)

FOR i=1 TO 6:OBJECT.X i,45*i+w4:OBJECT.Y i,h2: NEXT
OBJECT.ON
FOR i= 1 TO 6:OBJECT.VX i,1Ø*i:OBJECT.VY i,1Ø*i:NEXT
COLLISION ON
OBJECT.START
a=-1
WHILE a
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
COLLISION OFF
OBJECT.CLOSE
WINDOW CLOSE 2:SCREEN CLOSE 1
END

getroffen:
  'welches Objekt ist kollidiert?
  c=COLLISION(Ø)
  'noch Informationen in der Warteschlange?
  IF c=Ø THEN RETURN
  'mit welchem Objekt fand Kollision statt?
  o=COLLISION(c)
  IF o<Ø THEN  'Kollision mit Fensterrand
    IF o=-2 OR o=-4 THEN 'linker oder rechter Fensterrand
      cx=-(OBJECT.VX(c)) :OBJECT.VX c,cx
    ELSE                 'oberer oder unterer Fensterrand
      cy=-(OBJECT.VY(c)) :OBJECT.VY c,cy
```

```
    END IF
    OBJECT.START c
  ELSEIF o>Ø THEN         'Kollision mit anderem Objekt
      ox=-(OBJECT.VX(o)) :OBJECT.VX o,ox
      oy=-(OBJECT.VY(o)) :OBJECT.VY o,oy
      cx=-(OBJECT.VX(c)) :OBJECT.VX c,cx
      cy=-(OBJECT.VY(c)) :OBJECT.VY c,cy
      OBJECT.START o :OBJECT.START c
  END IF
GOTO getroffen
```

Der Programmbeginn bereitet Ihnen sicherlich keine Probleme. Nach der ON COLLISION-Anweisung werden die BOBs eingelesen und jeweils einmal kopiert. Nun folgen zwei Anweisungen, die zwar in diesem Programm überflüssig sind, in anderen Programmen aber durchaus wichtig sein können. Wenn es das Programm erfordert, daß die BOBs in einer bestimmten Reihenfolge übereinander erscheinen, so können Sie mit

```
OBJECT.PRIORITY Objektnummer,Priorität
```

diese Reihenfolge festlegen. Jedes Objekt mit einem höheren Wert für die Priorität erscheint über dem Objekt mit einem niedrigeren Wert für die Priorität. So lassen sich ganz einfach dreidimensionale Effekte verwirklichen.

Im Programm geht es mit den bereits besprochenen OBJECT.HIT-Anweisungen weiter. Die 6 BOBs werden auf Position gebracht, eingeschaltet, mit verschiedenen Geschwindigkeiten in X- und Y-Richtung versehen und gestartet. Damit auch jeder Crash festgestellt werden kann, wird vorher noch mit COLLISION ON die Unterbrechungsfähigkeit aktiviert.

Während die 6 BOBs ihre Bahnen zwischen den Fenstergrenzen ziehen, wartet das Programm auf einen Tastendruck des Anwenders. Gefällt diesem das lustige Hin- und Herspringen der Bälle nicht mehr, wird die Schleife verlassen. Nach dem COLLISION OFF (wichtig) wird keine Kollisions-Meldung mehr abgegeben. Bevor Window und Screen geschlossen werden, wird mit

```
OBJECT.CLOSE Objektnummer, Objektnummer,...
```

der von den BOBs belegte Speicherplatz wieder freigegeben. Die Freigabe kann für einzelne BOBs oder global für alle BOBs (wie im Programm geschehen) erfolgen. Wenn Sie diese Anweisung im Programm vergessen, brauchen Sie sich nicht zu wundern, wenn der verfügbare Speicher immer kleiner wird.

Werfen wir noch einen Blick auf die Kollisions-Abfrage in der Sub-Routine »getroffen«. Sie sehen, die ganze Routine läuft in einer Schleife. Diese Technik dürfte bei der

Kollisionsabfrage der günstigste Weg sein, um sicher zu gehen, daß auch das richtige Ereignis abgefragt wird und kein Ereignis mehr in der Kollisions-Warteschlange zurückbleibt. Eine Kollision kann durchaus mehrere Meldungen auslösen. Wird dann nur ein Ereignis aus der Schlange genommen, erhalten Sie beim nächsten Aufruf eine Meldung, die inzwischen schon Schnee von gestern ist.

Zuerst wird mit COLLISION(0) gefragt, welches Objekt die Kollisions-Meldung verursacht hat, und das Ergebnis in *c* festgehalten. An dieser Stelle kann die Schleife auch verlassen werden. Das Ergebnis *c* wird nun wieder in die COLLISION-Funktion eingesetzt. Jetzt erhalten wir den Gegner der Kollision und halten ihn in der Variablen *o* fest. Damit ist das Ereignis aus der Warteschlange gelöscht. Nun brauchen wir nur noch zu ermitteln, wer der Gegner des Zusammenstoßes war. Dies geschieht in dem IF-Block, der im Programm ausreichend kommentiert ist. Die Geschwindigkeiten werden aus den Funktionen OBJECT.VX und OBJECT.VY geholt und zur Fortsetzung der Objektbewegung negativ multipliziert. Dadurch prallen die Objekte voneinander bzw. vom Fensterrand ab.

10.2.2 Einfärben von BOBs

Damit haben Sie, bis auf einen, alle Objekt-Befehle kennengelernt. Dieser eine Befehl hat es allerdings in sich. Weder die Befehlsbeschreibung im Handbuch noch die in anderen Veröffentlichungen ist dazu angetan, das Verständnis für seine Anwendung zu wecken. Ich werde deshalb detaillierter als gewohnt darauf eingehen. Ich habe sogar extra ein Programm geschrieben, dessen einzige Aufgabe es ist, die Wirkungsweise dieser Anweisung verständlich zu machen.

Wie lautet denn nun dieser mysteriöse Befehl?

```
OBJECT.PLANES Objektnummer,Plane-Pick-Maske,Plane-On-Off-Maske
```

Die Anweisung ermöglicht es, BOBs aus der Kombination aller möglichen Bit-Ebenen einzufärben. Bei den Namen für die Masken habe ich absichtlich die englische Bezeichnung aufgeführt, da diese, wie Sie gleich feststellen werden, aussagefähiger ist als eine deutsche Übersetzung.

Wir stehen wieder einmal vor der Aufgabe, zwei Masken mit den richtigen Werten zu belegen. Die beiden Masken sind dieses Mal nur 8 Bits breit. Fangen wir mit der Plane-Pick-Maske an. Dazu rufen wir uns ins Gedächtnis zurück, was wir bisher über die Farbtiefe der BOBs in Screens von verschiedenen Tiefen gehört haben. Bei Vorgängen, die nicht so leicht zu durchschauen sind, soll man mit festen Parametern arbeiten. Stellen Sie sich dazu ein BOB mit einer Tiefe von 3 Ebenen (für 8 Farben) vor. Dieses BOB bringen Sie nun in einen Screen, der eine Tiefe von 5 Bit- Ebenen hat. Normalerweise werden für die Farben des BOBs die unteren 3 Bit-Ebenen von den 5 Ebenen der

Screen-Bitmap herangezogen. Soweit dürfte alles klar sein. Jetzt setzen wir die Anweisung OBJECT.PLANES ein. Diese Anweisung gestattet es, daß man sich die Ebenen, die man für die BOB-Farben benötigt, frei auswählen kann. Sie können sich für das BOB 3 beliebige Ebenen aus den 5 Ebenen des Screens herauspicken. Ist das nicht eine feine Sache? Den Wert für die Maske ermitteln wir auf die gewohnte Art und Weise. Sie haben sich zum Beispiel für die Ebenen 0, 2 und 4 entschieden. Wir rechnen also

```
2^0 + 2^2 + 2^4 = 21
```

Die Anweisung lautet also

```
OBJECT.PLANES Objektnummer,21
```

So weit so gut, doch für was soll die zweite Maske, Plane-On-Off, gut sein? (Sie können Sie bei der Anweisung auch weglassen.) Bleiben wir bei unserem Beispiel. Wir haben für das BOB die Ebenen 0, 2 und 4 herausgesucht. Die beiden Ebenen 1 und 3 werden für die Farbgebung nicht hinzugezogen. Schließlich haben wir ja nur ein BOB mit 3 Ebenen zur Verfügung. Das ist richtig und auch wieder nicht. Mit OBJECT.PLANES können wir nämlich auch bestimmen, was mit diesen beiden Ebenen bei der Farbgebung des BOBs geschehen soll! Sie können also beliebig festlegen, ob sie keine, eine der beiden, oder beide Ebenen einsetzen wollen. Wir entschließen uns in dem Beispiel, beide Ebenen 1 und 3 für die Farbgebung mitzuverwenden. Unsere Rechnung lautet wieder

```
2^1 + 2^3 = 10
```

Damit lautet die geänderte Anweisung von oben:

```
OBJECT.PLANES Objektnummer,21,10
```

Wenn man weiß, wie's funktioniert, ist es doch gar nicht so schwierig. Schauen Sie sich nun das versprochene Beispiel an, das die Anweisung mit seinen Parametern mit 40 verschiedenen Kombinationen von Bit-Ebenen demonstriert.

```
REM BobPlanes  Pfad: Darstellung/10Animation/BobPlanes
'P10-3
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):sw=320:tiefe=5
SCREEN 1,sw,sh,tiefe,1 :WINDOW 2,,,0,1
w=WINDOW(2) :h=WINDOW(3)
PALETTE 0,0,.6,0 :PALETTE 7,1,1,0
FOR i= 1 TO 31
  posi=8 :r=i
  LINE ((i-1)*10,10)-((i-1)*10+9,50),i,bf
  IF r-2^4>-1 THEN t-="1":r=i-2^4 ELSE t-="0"
```

```
  GOSUB zeig:IF r-2^3>-1 THEN t-="1":r=r-2^3 ELSE t-="Ø"
  GOSUB zeig:IF r-2^2>-1 THEN t-="1":r=r-2^2 ELSE t-="Ø"
  GOSUB zeig:IF r-2^1>-1 THEN t-="1":r=r-2^1 ELSE t-="Ø"
  GOSUB zeig:IF r-2^Ø>-1 THEN t-="1" ELSE t-="Ø"
  GOSUB zeig
NEXT

FOR i=1 TO 3
  OPEN ":Bild/BOB"+RIGHT-(STR-(i),1) FOR INPUT AS 1
  OBJECT.SHAPE i,INPUT-(LOF(1),1)
  CLOSE 1
NEXT
FOR i=1 TO 3:OBJECT.X i,45*i+2Ø:OBJECT.Y i,h-39: NEXT
OBJECT.ON
LOCATE 18,1:PRINT "weiter mit Leertaste"

'1Ø BitMap-Kombinationen fuer Plane-Pick
'4Ø BitMap-Kombinationen mit Plane-On-Off
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,Ø,Ø,1,1,1, j,k,Ø,Ø,Ø:taste:NEXT:NEXT:NEXT
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,Ø,1,1,1,Ø, j,Ø,Ø,Ø,k:taste:NEXT:NEXT:NEXT
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,1,1,1,Ø,Ø, Ø,Ø,Ø,j,k:taste:NEXT:NEXT:NEXT
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,1,Ø,1,1,Ø, Ø,j,Ø,Ø,k:taste:NEXT:NEXT:NEXT
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,1,Ø,Ø,1,1, Ø,j,k,Ø,Ø:taste:NEXT:NEXT:NEXT

FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,1,Ø,1,Ø,1, Ø,j,Ø,k,Ø:taste:NEXT:NEXT:NEXT
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,1,1,Ø,Ø,1, Ø,Ø,j,k,Ø:taste:NEXT:NEXT:NEXT
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,1,1,Ø,1,Ø, Ø,Ø,j,Ø,k:taste:NEXT:NEXT:NEXT
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,Ø,1,1,Ø,1, j,Ø,Ø,k,Ø:taste:NEXT:NEXT:NEXT
FOR j=Ø TO 1:FOR k=Ø TO 1
FOR i=1 TO 3:plan i,Ø,1,Ø,1,1, j,Ø,k,Ø,Ø:taste:NEXT:NEXT:NEXT
OBJECT.CLOSE
WINDOW CLOSE 2:SCREEN CLOSE 1
END
```

```
zeig:
  LOCATE posi:PRINT PTAB ((i-1)*10+1)t-
  posi=posi+1
RETURN

SUB plan (Objekt%,e0%,e1%,e2%,e3%,e4%,o0%,o1%,o2%,o3%,o4%) STATIC
  eben%= 2^0*e0% +2^1*e1% +2^2*e2% +2^3*e3% +2^4*e4%
  oo%= 2^0*o0% +2^1*o1% +2^2*o2% +2^3*o3% +2^4*o4%
  LOCATE 14,1:PRINT "PLANE PICK"e0% e1% e2% e3% e4% "Wert"eben%
  LOCATE 15,1:PRINT "PL. on-off"o0% o1% o2% o3% o4% "Wert"oo%
  LOCATE 16,1:PRINT "gilt fuer BOB"Objekt%
  OBJECT.PLANES Objekt%,eben%,oo%
END SUB

SUB taste STATIC
  WHILE INKEY-="" : WEND
END SUB
```

Bevor Sie das Programm starten, sollten Sie einen kurzen Blick auf den Programmablauf werfen. Zuerst wird ein Screen der Tiefe 5 eingerichtet und ein Fenster geöffnet. In der FOR/NEXT-Schleife werden die darstellbaren Farben gezeichnet und die gesetzten Bits der einzelnen Farben darunter geschrieben. Die drei BOBs werden eingelesen, positioniert und eingeschaltet. Es folgen zehn Bitmap-Kombinationen für Plane-Pick:

```
00111     10101
01110     11001
11100     11010
10110     01101
10011     01011
```

Zusammen mit den vier möglichen Varianten für Plane-On-Off

```
00     10
01     11
```

ergeben sich 40 verschiedene Farbkombinationen. Die Ausgabe auf den Bildschirm erfolgt im Unterprogramm »plan«. Die Übergabeparameter sind die einzelnen Bits der beiden Masken. Aus den einzelnen gesetzten Bits werden die Maskenwerte errechnet und in das Fenster geschrieben. Mit den errechneten Werten wird die Anweisung OBJEKT.PLANES nacheinander für die drei BOBs aufgerufen.

Wenn Sie das Programm starten, erscheinen oben am Bildschirm die 31 darstellbaren Farben. Die Hintergrundfarbe ist nicht berücksichtigt. Darunter finden Sie die gesetzten (1) oder nicht gesetzten (0) Bits der jeweiligen Farbe. Die unterste Reihe ist die Ebene 0, darüber befindet sich die Ebene 1 etc. Anschließend werden die einzelnen Bits

und der zugehörige Wert für Plane-Pick und für Plane-On-Off angezeigt. Die Werte gelten für das darunter geschriebene BOB.

Die Reihenfolge der Ausgaben geht folgendermaßen vor sich. Es erscheint eine Maske für Plane-Pick. Von der Maske On-Off ist kein Bit gesetzt. Bei jedem Tastendruck wird mit diesen Werten ein anderes BOB geändert. Nun werden die Masken-Bits von On-Off gesetzt, wieder nacheinander für die drei BOBs. Sind alle Varianten durchexerziert, folgt eine neue Maske für Plane-Pick etc. Die Farbänderung läßt sich an den drei BOBs gut erkennen. Aus den BOBs der drei runden Bälle sind ovale Football-Bälle geworden.

Schließlich wurden sie ursprünglich für einen Bildschirm von hoher Auflösung konstruiert. Damit Sie die Farbänderung an den BOBs noch besser verfolgen können, interessieren Sie sicherlich die Konstruktionswerte der drei BOBs. Die Größe ist einheitlich 40 (X) mal 20 (Y) Pixel. Sie sind in einer Tiefe von 3 Bit-Ebenen gezeichnet. Die Farbwerte der einzelnen Flächen betragen (von außen nach innen):

```
BOB 1: Farbe 1
BOB 2: Farbe 3 und 2
BOB 3: Farbe 4, 5, 6 und 7
```

Mit der Anweisung OBJEKT.PLANES haben Sie ein ideales Betätigungsfeld für Farbanimationen. Besonders viele Farbkombinationen erreichen Sie, wenn Sie BOBs von geringer Tiefe in Screens mit vielen Bitmaps einsetzen. Haben BOB und Screen die gleiche Anzahl an Bitmaps, so können Sie mit dieser Anweisung nur noch die Reihenfolge der Farben verändern. Da Sie das inzwischen längst wissen, kann ich Ihnen nur noch viel Spaß mit den Objekten allgemein und speziell mit farbanimierten BOBs wünschen.

10.2.3 Ändern der Eigenschaften

Bisher haben wir die Objekte, mit Ausnahme der BOB-Farben, so akzeptiert, wie sie der Objekt-Editor vergeben hat. Wollen Sie dem Objekt andere Eigenschaften mitgeben oder diese nachträglich verändern, so müssen Sie zuerst wissen, was für Möglichkeiten Ihnen zur Wahl stehen. Betrachten wir dazu den Aufbau der Zeichenkette mit den Daten, wie sie mit OBJECT.SHAPE übergeben werden.

Länge	**Bezeichnung**	**Offset**	**Standard-Werte**
Langwort	ColorsetOffset	0	0
Langwort	DataSetOffset	4	0
Langwort	Depth	8	
Langwort	Width	12	
Langwort	Height	16	
Wort	Flags	20	24 oder 25
Wort	PlanePick	22	2^Tiefe -1
Wort	PlaneOnOff	24	0
1. Wort	Grafik Ebene 0		
2. Wort	Grafik Ebene 0		
...			
letztes Wort	Grafik Ebene 0		
1. Wort	Grafik Ebene 1		
...			
letztes Wort	Grafik Ebene n		
Wort	Sprite Farbe 1		&h0ff weiß
Wort	Sprite Farbe 2		&h000 schwarz
Wort	Sprite Farbe 3		&hf80 orange

Flags			
VSprite	Bit 0	Wert 1	wenn Sprite
CollisionPlaneIncluded	Bit 1	Wert 2	
ImageShadowIncluded	Bit 2	Wert 4	
SAVEBACK	Bit 3	Wert 8	X
OVERLAY	Bit 4	Wert 16	X
SAVEBOB	Bit 5	Wert 32	

Was hier als Standard-Werte aufgeführt ist, wird vom Editor *ObjEdit* gesetzt. Besonders interessant sind die einzelnen Flags. Sicher haben Sie, zumindestens teilweise, von den einzelnen Begriffen schon gelesen oder gehört. Doch welche Eigenschaften verbergen sich eigentlich dahinter? Schauen wir uns die einzelnen Flags einmal an.

`VSprite`

Das Bit 0 wird bei einem Sprite gesetzt und bei einem BOB gelöscht. Dieses Flag ist damit die Grundlage für die anderen Flags. Ist es ein Sprite, werden die Farben aus den letzen drei Daten-Worten des Strings geholt. Bei einem Bob sind diese drei Worte Bestandteil der obersten Bitplane.

`CollisionPlaneInclude`
Wenn dieses Flag nicht gesetzt ist, so werden alle gesetzten Bits der Objekt-Daten zur Erkennung einer Kollision herangezogen. Wenn Sie allerdings dieses Flag setzen, so können Sie eine eigene Ebene (Maske) zeichnen, die dann die Berührung mit einem anderen Objekt registriert. So können Sie zum Beispiel in der Mitte des Objektes eine kleine Fläche maskieren (zeichnen). Eine Kollision wird dann erst registriert, wenn das andere Objekt in das maskierte Objekt eingedrungen ist.

`ImageShadowIncluded`
Auch dieses Flag steht wie das Kollisions-Flag für eine eigene Bit-Ebene. Auch hier können Sie eine Maske zeichnen.(Die Ebenen müssen bei beiden Flags die Größe einer Objekt-Ebene haben.) Jeder Punkt der Maske bedeutet, daß nur diese Punkte der Objekt-Grafik zu sehen sind. Obwohl damit unter Umständen nur ein Teil des Objektes zu sehen ist, gilt für die Kollisionserkennung das vollständige Objekt oder die Kollisions-Maske. Zusammen mit anderen Flags ergeben sich andere Varianten. Löschen Sie zum Beispiel das OVERLAY-Flag, so bleibt die Grafik als OVERLAY-Bild bestehen.

`SAVEBACK`
Der Bildhintergrund wird zwischengespeichert, damit er wieder hergestellt werden kann, wenn das BOB weiterbewegt wurde. Leider führt dieses Flag zum Flimmern vor allem von größeren BOBs. Bei gelöschtem Flag wird also der Bildhintergrund nicht gerettet. Der BOB wirkt wie ein Pinsel. Wenn Sie in Ihren Programmen nur statische BOBs einsetzen, löschen Sie dieses Flag. Damit vermeiden Sie das Flimmern des BOBs.

`OVERLAY`
Die Farbe Null ist bei den BOBs immer transparent. Bei nicht gesetztem OVERLAY-Flag verdeckt aber auch die Hintergrundfarbe des Objektes den Bildhintergrund.

`SAVEBOB`
Dieses Flag ist das Gegenteil von SAVEBACK. Unter Basic können wir dieses Flag unberücksichtigt lassen.

Auf Anhieb ist die Funktion der einzelnen Flags sicherlich nicht zu durchschauen. Das macht aber nicht viel aus. Wie oft in solchen Fällen zeigt die praktische Anwendung mehr als viele Worte. Sie finden daher anschließend ein Programm, das Ihnen die Funktion der besprochenen Flags demonstriert. Sie können Ihren BOBs auch eine eigene Schattenebene und/oder Kollisions-Maske hinzufügen. Die Farbzusammenstellung der BOB-Farben läßt sich problemlos ändern. Aber damit nicht genug, auch die Farben der Sprites können Sie nach Belieben festlegen. Damit jede Änderung sofort sichtbar wird, ist ein eigenes Demonstrations-Programm eingebaut. Bevor ich jedoch auf weitere Einzelheiten eingehe, schauen Sie sich am besten das Super-Programm erst einmal an.

```
REM OBJECTdevil  Pfad: Darstellung/1ØAnimation/OBJEKTdevil
'P1Ø-4
IF FRE(-1)<11ØØØØ& THEN PRINT "Speicher zu klein":BEEP:END
ON ERROR GOTO Fehler
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
scr=Ø:aktiv=Ø:wi=1

start:
  frei
  LOCATE 17,1:INPUT "bitte Objektnamen";namen-
  IF namen-="" THEN start

start2:
  IF aktiv THEN GOSUB klaeren
  GOSUB ObjektEinlesen :aktiv=-1
  FOR i=Ø TO 1ØØØ:NEXT
  PlusPlane=Ø
  IF scr THEN WINDOW CLOSE wi:SCREEN CLOSE 1:scr=Ø
  CLS
  wi=wi+1
  IF tif&>2 THEN
    IF tif&>4 THEN
      sw=32Ø:mo=1
    ELSE
      sw=64Ø:mo=2
    END IF
    scr=-1:SCREEN 1,sw,sh,tif&,mo :WINDOW wi,,,Ø,1
  END IF
  w=WINDOW(2) :w2=w/2+37:h=WINDOW(3):h2=h/2
  PALETTE Ø,.4,.4,.4
  LOCATE 17,1:PRINT "Objektname:       ";namen-
  GOSUB FlagsErmitteln
  GOSUB ObjektZeigen
  GOSUB ObjektdatenZeigen
  GOSUB FlagsZeigen
  GOSUB Auswahl
tast: ta-=INKEY-:IF ta-=""THEN tast
  IF ta-<CHR-(129) OR ta->CHR-(137) THEN BEEP:GOTO tast
  wahl=ASC(ta-)-128
  ON wahl GOTO KE,SE,SBack,OV,SBob,PPOO,SF,start,ende
```

```
ende:
  GOSUB klaeren
  IF scr THEN WINDOW CLOSE wi:SCREEN CLOSE 1
  CLS
  IF fehl THEN
    BEEP:ON fehl GOSUB f1,f2,f3
  END IF
END

f1: PRINT "Objekt nicht gefunden":RETURN
f2: PRINT "Speicher reicht nicht aus!":RETURN
f3: PRINT "Fehler Nummer: "feNu:RETURN

Fehler:
  feNu=ERR:fehl=3
  IF feNu=53 THEN fehl=1
  IF feNu=7 THEN fehl=2
RESUME ende

klaeren:
  IF aktiv = Ø THEN RETURN
  OBJECT.OFF
  IF sprite = Ø THEN OBJECT.CLOSE
  ERASE obj :aktiv=Ø
RETURN

ObjektEinlesen:
  OPEN namen- FOR INPUT AS 1
    nu= LOF(1)/2:nu=nu-5:DIM obj(nu)
    FOR i=1 TO 5:obj(i)=CVL(INPUT-(4,1)):NEXT
    FOR i=6 TO nu:obj(i)= CVI(INPUT-(2,1)):NEXT 'Daten
  CLOSE #1
    cs&=obj(1)             'ColorSet
    ds&=obj(2)             'DataSet
    tif&=obj(3)            'Tiefe
    br&=obj(4)             'Breite
    ho&=obj(5)             'Hoehe
    fl =obj(6)             'Flags
    pp =obj(7)             'PlanePick
    poo=obj(8)             'PlaneOnOff
    c1 =obj(nu-2)          'Spr.Farbe1
    c2 =obj(nu-1)          'Spr.Farbe2
    c3 =obj(nu)            'Spr.Farbe3
RETURN
```

```
ObjektZeigen:
  ON COLLISION GOSUB getroffen
  GOSUB DemoGrafik
  OPEN "I",1,namen-
    OBJECT.SHAPE 1,INPUT-(LOF(1),1)
  CLOSE #1
  OBJECT.SHAPE 2,1
  OBJECT.X 1,0: OBJECT.Y 1,155
  OBJECT.X 2,w-br&-10 : OBJECT.Y 2,155
  OBJECT.ON :COLLISION ON
  OBJECT.VX 1,40:OBJECT.START
  a=-1
  WHILE a
    SLEEP
  WEND
  OBJECT.STOP :COLLISION OFF
  GOSUB DemoGrafik
RETURN

getroffen:
  c=COLLISION(0)
  IF c=0 THEN RETURN
  o=COLLISION(c)
  IF o>0 THEN a=0
GOTO getroffen

DemoGrafik:
  FOR i=1 TO 2^tif& -1
    FOR j=0 TO 2
      AREA (CINT(RND*(w-2)),CINT(RND*(h-147)+145))
    NEXT
    COLOR i:AREAFILL
  NEXT
  COLOR 1
RETURN

FlagsErmitteln:
  sprite =0  :Kollision=0 :Schatten=0
  saveback=0 :Overlay=0   :SaveBob=0
  IF fl AND 1 THEN sprite =-1:typ-="Sprite" ELSE typ-="BOB"
  IF fl AND 2 THEN Kollision=-1
  IF fl AND 4 THEN Schatten=-1
```

```
  IF fl AND 8 THEN saveback=-1
  IF fl AND 16 THEN Overlay=-1
  IF fl AND 32 THEN SaveBob=-1
RETURN

ObjektdatenZeigen:
  LOCATE 1 :PRINT PTAB(w2)"OBJEKTDATEN"
  LOCATE 3 :PRINT PTAB(w2)"Typ:     "typ-
  LOCATE 4 :PRINT PTAB(w2)"ColorSet   "cs&
  LOCATE 5 :PRINT PTAB(w2)"DataSet    "ds&
  LOCATE 6 :PRINT PTAB(w2)"Tiefe      "tif&
  LOCATE 7 :PRINT PTAB(w2)"Breite     "br&
  LOCATE 8 :PRINT PTAB(w2)"Hoehe      "ho&
  LOCATE 9 :PRINT PTAB(w2)"Flags      "fl
  LOCATE 10 :PRINT PTAB(w2)"PlanePick  "pp
  LOCATE 11 :PRINT PTAB(w2)"PlaneOnOff "poo
  IF sprite THEN
    LOCATE 12 :PRINT PTAB(w2)"Spritefar1 "HEX-(c1)
    LOCATE 13 :PRINT PTAB(w2)"Spritefar2 "HEX-(c2)
    LOCATE 14 :PRINT PTAB(w2)"Spritefar3 "HEX-(c3);
  END IF
RETURN

FlagsZeigen:
  LOCATE 10,1 :PRINT "FLAGS:"
  LOCATE 10,7 :PRINT "VSprite ...... "sprite
  LOCATE 11,7 :PRINT "Collision Incl."Kollision
  LOCATE 12,7 :PRINT "ImageShad.Incl."Schatten
  LOCATE 13,7 :PRINT "SAVEBACK ..... "saveback
  LOCATE 14,7 :PRINT "OVERLAY ...... "Overlay
  LOCATE 15,7 :PRINT "SAVEBOB ...... "SaveBob
RETURN

Auswahl:
  COLOR 3
  LOCATE 1,1  :PRINT "F1=Kollisions-Eb. hinzu"
  LOCATE 2,1  :PRINT "F2=Schatten-Ebene hinzu"
  LOCATE 3,1  :PRINT "F3=SAVEBACK-Flag +/-"
  LOCATE 4,1  :PRINT "F4=OVERLAY-Flag +/-"
  LOCATE 5,1  :PRINT "F5=SAVEBOB-Flag +/-"
  LOCATE 6,1  :PRINT "F6=PlanePick OnOff neu"
  LOCATE 7,1  :PRINT "F7=Spritefarben "
```

```
  LOCATE 8,1  :PRINT "F8=neues Objekt"
  LOCATE 9,1  :PRINT "F9=ENDE"
  COLOR 1
RETURN

Fehler1:ftext-="Flag bei Sprites nicht moeglich":GOTO Fausg
Fehler2:ftext-="Option nur fuer Spritefarben  ":GOTO Fausg
Fehler4:ftext-="Ebene existiert bereits!      ":GOTO Fausg
Fehler5:ftext-="falsche Reihenfolge shadow/col":GOTO Fausg

Fausg:
  COLOR 3
  LOCATE 18,1:PRINT ftext-
  COLOR 1
GOTO tast

KE:
'Kollisions-Ebene hinzufuegen
  'Aufbau der BOB-Ebenen
  'erste Bit-Plane bis
  'letzte BitPlane
  'ImageShadow BitPlane
  'Collision BitPlane
  'Bitte achten Sie auf die richtige Reihenfolge
  IF sprite THEN Fehler1
  IF Kollision THEN Fehler4
  flwert=2
GOTO zeichnen

SE:
'Schatten-Ebene hinzufuegen
  IF sprite THEN Fehler1
  IF Schatten THEN Fehler4
  IF Kollision THEN Fehler5
  flwert=4
GOTO zeichnen

zeichnen:
  OBJECT.OFF
  OBJECT.CLOSE
  IF scr THEN WINDOW CLOSE wi:SCREEN CLOSE 1:scr=Ø
  wi=wi+1
  scr=-1:SCREEN 1,32Ø,sh,1,1 :WINDOW wi,,,Ø,1
```

```
ZeichStart:
  LINE (Ø,Ø)-(br&+4,ho&+2),1,bf
  LINE (Ø,Ø)-(br&-1,ho&-1),Ø,bf
  LOCATE 17,1:PRINT "F1=neue Zeichnung"
  LOCATE 18,1:PRINT "F2=speichern"
  LOCATE 2Ø,1:PRINT "In das umrandete Feld mit "
  LOCATE 21,1:PRINT "linker Maustaste zeichnen"
  a=-1:b=Ø
  WHILE a
    muss=MOUSE(Ø)
    WHILE MOUSE(Ø)><Ø
      PSET (MOUSE(1),MOUSE(2))
    WEND
    ta-=INKEY-
    IF ta-=CHR-(129) THEN a=Ø:b=-1
    IF ta-=CHR-(13Ø) THEN a=Ø
  WEND
  IF b THEN ZeichStart
  anz=3+(ho&)*INT((br&-1+16)/16)
  DIM bild(anz)
  GET (Ø,Ø)-(br&-1,ho&-1),bild
  PlusPlane=-1
  fl=fl OR flwert
  obj(6)=fl
  GOSUB StringZusammensetzen
  ERASE bild
  GOSUB klaeren
GOTO start2

StringZusammensetzen:
  IF RIGHT-(namen-,6)<>".devil" THEN namen-=namen-+".devil"
  OPEN namen- FOR OUTPUT AS 1
    FOR i=1 TO 5
      PRINT #1,MKL-(obj(i));
    NEXT
    FOR i=6 TO nu
      PRINT #1,MKI-(obj(i));
    NEXT
    IF PlusPlane THEN
      FOR i=3 TO anz-1
        PRINT #1,MKI-(bild(i));
```

```
      NEXT
    END IF
  CLOSE #1
RETURN

SBack:
'SAVEBACK-Flag setzen/loeschen
  flist=Ø:IF saveback THEN flist=-1
  flwert=8:GOTO FlagSetzen

OV:
'OVERLAY-Flag setzen/loeschen
  flist=Ø:IF Overlay THEN flist=-1
  flwert=16:GOTO FlagSetzen

SBob:
'SAVEBOB-Flag setzen/loeschen
  flist=Ø:IF SaveBob THEN flist=-1
  flwert=32:GOTO FlagSetzen

FlagSetzen:
  IF sprite THEN Fehler1
  IF flist THEN
    fl=fl AND NOT flwert
  ELSE
    fl=fl OR flwert
  END IF
  obj(6)=fl
  GOSUB StringZusammensetzen
  GOSUB klaeren
GOTO start2

PPOO:
'PlanePick PlaneOnOff neu
  IF sprite THEN Fehler1
  frei
  LOCATE 17,1:PRINT "ACHTUNG! Keine Pruefung der Eingabe!"
  LOCATE 18,1:INPUT "PlanePick,PlaneOnOff";pp,poo
  obj(7)=pp
  obj(8)=poo
  GOSUB StringZusammensetzen
  GOSUB klaeren
GOTO start2
```

```
SF:
'Sprite-Farben aendern
  IF sprite = Ø THEN Fehler2
  FOR j = Ø TO 2
    far=Ø:frei
    LOCATE 17,1:PRINT "Zusammensetzung Sprite-Farbe"3-j
    LOCATE 18,1:
    INPUT "rgb z.B. 15,Ø,1Ø ";Fa(2),Fa(1),Fa(Ø)
    FOR i=Ø TO 2
      IF Fa(i)>15 THEN Fa(i)=15
      far=far+ABS(Fa(i))*2^(i*4)
    NEXT
    obj(nu-j)=far
  NEXT
  GOSUB StringZusammensetzen
  GOSUB klaeren
GOTO start2

SUB frei STATIC
  LOCATE 17,1:PRINT SPACE-(4Ø)
  LOCATE 18,1:PRINT SPACE-(4Ø)
END SUB
```

Da das Programm dazu gedacht ist, daß Sie Ihre Objekte nach Ihren Wünschen ändern und ergänzen können, gehen wir auf den Programmaufbau nicht näher ein. Die wichtigen Stellen sind im Programm kommentiert, die Labels und die Variablen tragen aussagefähige Namen. Es wird Ihnen daher nicht schwer fallen, die einzelnen Programmschritte zu verfolgen.

Das Programm wartet nach dem Start auf die Eingabe eines Objekt-Namens. Geben Sie bitte den vollständigen Pfad ein. Zum Beispiel

```
:Bild/Bob1
```

Nun werden die Daten des Objektes eingelesen. Anhand der Tiefe des Objektes wird die Tiefe des Screens überprüft und gegebenenfalls eine neuer Bildschirm mit Fenster geöffnet. Um die Auswirkung des Objektes, je nach den gesetzten Flags, auf den Bildhintergrund zu zeigen, werden einige verschiedenfarbige Dreiecke gezeichnet. Das Objekt erscheint in doppelter Ausfertigung. Einmal ist es am rechten Bildrand postiert, und das zweite Mal bewegt es sich von der linken Bildschirmkante auf das erste Objekt zu. Zur Demonstration der Kollisions-Maske wird die Bewegung durch eine Kollisions-Meldung unterbrochen.

Anschließend werden auf der rechten Bildhälfte die Objektdaten gezeigt. Die Flags werden ausgelesen und auf der linken Bildschirmhälfte ausgegeben. In roter Schrift sind, ebenfalls auf der linken Bildschirmseite, die einzelnen Programm-Optionen zu sehen. Die Auswahl der Optionen erfolgt durch die Funktionstasten F1 bis F9. Durch F1 und F2 können Sie dem BOB eine Kollisions- oder Schatten-Maske hinzufügen. Es öffnet sich eine neuer Bildschirm mit der Tiefe 1. Da nur eine Bit-Ebene hinzugefügt wird, ist das ausreichend. In der linken oberen Bildschirmecke wird ein rechteckiger Bereich markiert, in dem die Maske gezeichnet werden kann. Zum Zeichnen mit der linken Maustaste wendet das Programm nur PSET an. An dieser Stelle können Sie weitere Zeichenroutinen einfügen. Bei mißglückter Grafik kann die Zeichnung mit F1 neu begonnen werden. Mit F2 wird das BOB mit der zusätzlichen Ebene abgespeichert. Als Name wird der File-Name mit dem Zusatz *.devil* versehen.

Wie bei jeder Änderung erscheint anschließend, wie zu Beginn geschildert, das geänderte Objekt. Dabei läuft wieder das kleine Demonstrationsprogramm ab. Damit können Sie Ihre Änderung sofort kontrollieren. Nun kann die nächste Änderung durchgeführt werden. Der Name ändert sich dabei nicht mehr, damit nicht plötzlich die Diskette wegen lauter neuer Objekt-Dateien überläuft. Wollen Sie das Objekt in dem geänderten Zustand behalten, empfiehlt es sich daher, den Namen von der Workbench (oder vom CLI) aus zu ändern.

Die Funktionstasten F2 bis F5 löschen die Flags SAVEBACK, OVERLAY oder SAVEBOB oder fügen die Flags hinzu. Die geänderten Flags können Sie anschließend an den ausgegebenen Daten und natürlich am Objekt selbst verfolgen. F6 ändert die BOB-Ebenen für die Farbgebung durch PlanePick und PlaneOnOff, wie wir es bereits besprochen haben. Die Taste F7 wird zur Änderung der Spritefarben gedrückt. Die Sprite-Farben belegen die letzten drei Worte des Daten-Strings. Mit F8 können Sie ein neues Objekt einlesen und mit F9 beenden Sie das Programm. Sie sehen, Sie können allerhand mit Ihren Objekten anstellen. Probieren Sie die einzelnen Optionen in aller Ruhe an Ihren Lieblings-Objekten aus.

10.3 Simple Sprites

Wenn Sie mehr über die Objekte kennenlernen wollen, müssen Sie wieder die System-Routinen bemühen. Als Hardware-Sprites, die auch Simple Sprites genannt werden, stehen uns maximal 8 Sprites, mit den Nummern 0 bis 7, zur Verfügung. Für die Breite (max. 16 Pixel) und die Höhe gelten die bekannten Beschränkungen. Bei der Farbgebung werden die oberen 16 Farben der ColorMap herangezogen. Dabei erhalten immer 2 Sprites die gleichen Farbregister zugewiesen:

Die Sprites 0 und 1 erhalten die Farbnummern 16 bis 19.
Die Sprites 2 und 3 erhalten die Farbnummern 20 bis 23.

Die Sprites 4 und 5 erhalten die Farbnummern 24 bis 27.
Die Sprites 6 und 7 erhalten die Farbnummern 28 bis 31.

Die jeweils unterste Farbe der Sprites wird transparent ausgegeben. Zur Handhabung der Hardware-Sprites genügen vier Routinen der Grafik-Library. Die Programmierung ist daher nicht aufwendig. Zuerst borgen Sie sich vom System ein Sprite aus:

Format: `GetSprite(sprite,pick)`

Die Funktion gibt einen Wert von –1 zurück, wenn das Sprite nicht ordnungsgemäß übergeben werden konnte. Mit *pick* können Sie entweder bestimmen, ob Sie ein bestimmtes Sprite haben wollen, oder Sie können die Auswahl dem System überlassen. Im ersten Fall tragen Sie einfach die Nummer des gewünschten Sprites als *pick* ein. Soll das System die Auswahl vornehmen, so geben Sie als *pick* den Wert –1 ein. Der Parameter *sprite* zeigt auf eine SimpleSprite-Struktur, die Sie vorher erstellen müssen. Dazu belegen Sie einen Speicherbereich für eine Struktur von 10 Worten Länge. In die vierte Position des Speicherbereiches tragen Sie in Wortlänge die Höhe des Sprites in Pixel ein. Das ist schon alles. Den Rest der Eintragungen besorgt das System.

Nun weiß das System, welche Nummer Ihr neues Sprite hat. Als nächstes sollten Sie dem System erklären, wie Ihr Sprite auszusehen hat:

Format: `ChangeSprite(vp,s,newdata)`

Der Parameter *vp* zeigt auf den ViewPort, in dem das Sprite ausgegeben werden soll. Mit einem Wert von 0 gilt der aktuelle ViewPort. *s* zeigt auf die SimpleSprite-Struktur. Als *newdata* können Sie endlich einen Zeiger auf die Sprite-Daten eingeben. Richtiger gesagt zeigen Sie damit auf eine Struktur mit dem Namen UserSpriteData. In dieser Struktur werden die ersten beiden Worte Null gesetzt, da sie vom System belegt werden. Es folgen die Sprite-Daten in Wortlänge. Zuerst kommt die erste Daten-Zeile der Bit-Ebene 0, dann die Zeile der Bit-Ebene 1, dann die zweite Zeile etc. Abgeschlossen wird die Struktur durch zwei Worte mit dem Wert Null. Diese Library-Routine können Sie nicht nur dazu einsetzen, um die Sprite-Daten zu Programmbeginn festzulegen. Auch während des Programmablaufes können Sie damit andere Daten für das Sprite bestimmen. Damit können Sie aus den statischen Sprites bewegte Ojekte zaubern!

Mit den ersten beiden Routinen bringen Sie das Sprite bereits auf den Bildschirm. Damit es nicht länger dort herumsteht, sollten Sie ihm etwas Bewegung verschaffen:

Format: MoveSprite(vp,sprite,x,y)

Die beiden Parameter *vp* und *sprite* gelten wieder für den ViewPort und die Simple-Sprite-Struktur. Für *x* und *y* geben Sie die Position an, zu der sich das Sprite bewegen soll. Dabei ist zu beachten, daß auch bei einem Hires-Schirm die X-Koordinate wie für einen Lores-Schirm angegeben werden muß. Die X-Position 300 auf einem Screen von 640 Pixel Breite setzt das Sprite auf die Position 600. Die letzte Routine sorgt für die Rückgabe des ausgeborgten Sprites an das System:

Format: FreeSprite(num)

Der Parameter *num* ist die Nummer des Sprites, das ans System zurückgegeben werden soll. Wenn Sie in Ihrem Programm das oder die Sprites nicht zurückgeben, sind sie für weitere Anwendungen verloren. Sie merken das daran, daß sie sogar auf der Workbench immer noch vorhanden sind. Da hilft dann nur noch der Neustart des Systems.

Damit wissen Sie bereits alles, um Hardware-Sprites programmieren zu können. Ein kleines Beispiel zeigt Ihnen wieder die praktische Anwendung der einzelnen Routinen.

```
REM Seilbahn  Pfad: Darstellung/1ØAnimation/Seilbahn
'P1Ø-5
CLEAR
DEFINT a-z
GOSUB LibraryOeffnen
GOSUB SpeicherReservieren :IF fehl THEN ende
PALETTE Ø,1,Ø,Ø:PALETTE 17,Ø,Ø,Ø
PALETTE 2,.4,.4,1:PALETTE 3,.6,.6,.6
FOR i = Ø TO 99 STEP 2
  READ sd%:POKEW d1&+i,sd% AND 65535&
  POKEW d2&+i,sd% AND 65535&
NEXT

POKEW s1&+4,23      'SpriteHoehe Sprite1
POKEW s2&+4,23      'SpriteHoehe Sprite2
num(1)=GetSprite&(s1&,-1):IF num(1)=-1 THEN fehl=2:GOTO ende
num(2)=GetSprite&(s2&,-1):IF num(2)=-1 THEN fehl=2:GOTO ende
CALL ChangeSprite&(Ø,s1&,d1&)
CALL ChangeSprite&(Ø,s2&,d2&)
```

```
start:
  CLS:x=3Ø:y=145:GOSUB Mast :x=59Ø:y=5:GOSUB Mast
  LINE (3Ø,15Ø)-(59Ø,1Ø),3
  x1=3Ø :y1=147:x2=264:y2=3Ø
  CALL MoveSprite&(Ø,s1&,x1,y1)
  CALL MoveSprite&(Ø,s2&,x2,y2)
  LOCATE 22,35:PRINT "anfangen = Leertaste":taste
  LOCATE 19,35:PRINT "Wenn die Gondeln zu schnell werden,"
  LOCATE 2Ø,35:PRINT "koennen sie abstuerzen!"
  LOCATE 22,35:PRINT "anhalten = Leertaste"

  t=3ØØ :b=-1 :pu=Ø:crash=Ø :puMAX=INT(RND*5)*2+5Ø
  WHILE b
    t=t-1Ø :x1=3Ø :y1=147:x2=264:y2=3Ø:a=-1
    WHILE a
      x1=x1+2:y1=y1-1:IF y1<3Ø THEN a=Ø
      x2=x2-2:y2=y2+1
      CALL MoveSprite&(Ø,s1&,x1,y1)
      CALL MoveSprite&(Ø,s2&,x2,y2)
      FOR n=Ø TO t:NEXT
    WEND
    pu=pu+1:a=-1
    WHILE a
      x1=x1-2:y1=y1+1
      x2=x2+2:y2=y2-1:IF y2<3Ø THEN a=Ø
      CALL MoveSprite&(Ø,s1&,x1,y1)
      CALL MoveSprite&(Ø,s2&,x2,y2)
      FOR n=Ø TO t :NEXT
    WEND
    pu=pu+1 :IF INKEY-<>"" THEN b=Ø
    IF pu>puMAX THEN b=Ø:pu=Ø:crash=-1
  WEND
  IF crash THEN
    WHILE x2<35Ø
      x1=x1-2:y1=y1+1:x2=x2+2:y2=y2-1
      CALL MoveSprite&(Ø,s1&,x1,y1)
      CALL MoveSprite&(Ø,s2&,x2,y2)
    WEND
  END IF
LOCATE 1,1:PRINT "Sie erreichten: "pu" Punkte"
IF pu=puMAX THEN LOCATE 2,1:PRINT "damit haben Sie das Spiel
gewonnen"
```

```
LOCATE 3,1:PRINT "maximale Punktzahl="puMAX" Punkte"
LOCATE 5,1:PRINT "Noch ein Spiel? j/n"
warte: ta–=INKEY–:IF ta–="" THEN warte
IF ta–="j" OR ta–="J" THEN start

ende:
  FOR i=1 TO 2
    IF num(i)>Ø THEN CALL FreeSprite&(num(i))
  NEXT
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN
    ON fehl GOSUB F1,F2
    BEEP:PRINT ft–
  END IF
  LIBRARY CLOSE
END

F1:ft–= "Speicher nicht ausreichend":RETURN
F2:ft–= "kein freier Sprite":RETURN

LibraryOeffnen:
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION GetSprite&() LIBRARY
LIBRARY ":bue/intuition.library"
LIBRARY ":bue/graphics.library"
RETURN

SpeicherReservieren:
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  s1&=AllocRemember&(rk&,16,art&)   'fuer SpriteStruktur1
    IF s1&=Ø THEN fehl=1:RETURN
  s2&=AllocRemember&(rk&,16,art&)   'fuer SpriteStruktur2
    IF s2&=Ø THEN fehl=1:RETURN
  d1&=AllocRemember&(rk&,1ØØ,art&)  'fuer SpriteDaten1
    IF d1&=Ø THEN fehl=1:RETURN
  d2&=AllocRemember&(rk&,1ØØ,art&)  'fuer SpriteDaten2
    IF d2&=Ø THEN fehl=1
RETURN

Mast:
  COLOR 2
  FOR y1= Ø TO 4Ø STEP 3
    x1=y1/4+1: LINE (x-x1,y+y1)-(x+x1,y+y1)
```

```
  NEXT
  COLOR 1
RETURN
SUB taste STATIC
  WHILE INKEY-="":WEND
END SUB

Gondel:
DATA &h0000,&h0000
DATA &h0000,&h0180, &h0000,&h03c0, &h0180,&h03c0
DATA &h0180,&h0180, &h0180,&h0180, &h0180,&h0180
DATA &h0180,&h0180, &h0180,&h0180, &h0180,&h0180
DATA &hffff,&hffff, &h8811,&h8811, &h9009,&h9009
DATA &ha005,&ha005, &hc003,&hc003, &h8001,&h8001
DATA &hffff,&h0000, &hffff,&h0000, &hffff,&h0000
DATA &hffff,&h0000, &hffff,&h0000, &hffff,&h0000
DATA &hffff,&h0000, &hffff,&h0000
DATA &h0000,&h0000
```

Gleich nach dem Öffnen der Library und den Speicherreservierungen werden die Daten der beiden Sprites eingelesen und in den Speicher geschrieben. Anschließend wird die Höhe der Sprites in die SimpleSprite-Strukturen eingetragen. Damit können bereits mit *GetSprite* die beiden Sprites vom System ausgeliehen werden. Nun werden mit *Change-Sprite* die Sprite-Daten übergeben.

Beim Label »start« werden zwei Masten mit einem Drahtseil dazwischen gezeichnet. Die beiden Gondeln bzw. Sprites werden in den beiden Schleifen so positioniert, daß die Seilrolle der Gondeln immer auf dem Seil zu liegen kommt. Wie bereits erwähnt, sind die X-Positionen nur mit den halben Werten der Screen-Position aufgeführt.

Die äußere WHILE/WEND-Schleife ist für die Anzahl der Durchgänge verantwortlich. Mit jedem Durchgang verkürzt sich die Zeitschleife, die zwischen den einzelnen Bewegungen der Sprites liegt. Der Spieler hat die Aufgabe, die Gondeln rechtzeitig anzuhalten, bevor sie vom Drahtseil stürzen. Schafft er das im letzten Augenblick, so erreicht er die maximale Punktzahl.

Zum Schluß werden mit *FreeSprite* die Sprites wieder zurückgegeben. Nun brauchen nur noch die reservierten Speicherbereiche freigegeben und die Library geschlossen zu werden.

Gegenüber den Sprites des normalen Basic müssen Sie bei den Simple Sprites in punkto Komfort einige Abstriche machen. Daß die Programmierung problemlos ist, davon konnten Sie sich selbst überzeugen. Wenn Sie es geschickt anstellen, können Sie damit sogar bewegte Sprites programmieren. Probieren Sie es ruhig einmal aus.

10.3.1 Multicolor-Sprites

Normalerweise kann ein Sprite in vier Farben dargestellt werden. Ziehen wir die transparente Hintergrundfarbe ab, verbleiben nur noch 3 Farben. Das ist nicht gerade berauschend. Liebhaber farbenfroher Sprites brauchen jedoch nicht zu resignieren. Es gibt eine Möglichkeit, 16-farbig Sprites zu erstellen. Ziehen wir davon wieder die transparente Hintergrundfarbe ab, so verbleiben immer noch 15 Farben. Damit können Sie, gegenüber einem normalen Sprite, fünfmal soviel Farben unterbringen!

Allerdings müssen Sie sich dabei Einschränkungen in der Anzahl darstellbarer Sprites gefallen lassen. Ein Multicolor-Sprite ist nichts anderes als zwei einzelne Sprites, die miteinander verknüpft werden. Damit können Sie maximal 4 solcher Sprites gleichzeitig darstellen. Dabei gehören die Sprites 0 und 1, 2 und 3 etc. zusammen. Die Farben stammen aus den oberen 16 Farben (16–31) der ColorMap.

Im letzten Kapitel haben Sie gelesen, daß in der Struktur UserSpriteData die ersten beiden Worte mit Null vorgegeben werden, da sie vom System belegt werden. Dabei erhält das erste Kontroll-Wort die Startposition des Sprites in horizontaler und in vertikaler Richtung, also die Koordinaten, an denen die erste Zeile des Sprites gezeigt wird. Das zweite Kontrollwort legt hauptsächlich die Höhe des Sprites (Stopposition) fest. Außerdem gibt es da noch, und da wird es für uns interessant, das ominöse 7. Bit, genannt ATTACHED-Bit.

Das Rezept zum Backen eines Multicolor-Sprites lautet daher: Man nehme zwei zusammengehörende Sprites von gleicher Größe, setze in dem zweiten Kontrollwort des zweiten Sprites (immer ungerade Spritenummer) das siebente Bit, bringe beide Sprites auf eine Position, und fertig ist das 16-farbige Sprite. Die 16 Farben resultieren natürlich aus den insgesamt 4 Bitmaps der beiden Sprites. Die Konstruktion eines solchen zusammengehörenden Doppel-Sprites ist nicht ganz einfach. Beim Zeichnen der 4 Bitmaps muß man da schon gehörig aufpassen, um den Grafikpunkt mit der richtigen Farbe auf die 4 Bit-Ebenen zu verteilen. Die folgende Tabelle soll Ihnen bei der Farbgebung eine kleine Hilfe sein:

Farbnummer	**erstes Sprite**		**zweites Sprite**	
der Farbtafel	**Ebene 0**	**Ebene 1**	**Ebene 2**	**Ebene 3**
16	transparent			
17	x			
18		x		
19	x	x		
20			x	
21	x		x	
22		x	x	
23	x	x	x	
24				x
25	x			x
26		x		x
27	x	x		x
28			x	x
29	x		x	x
30		x	x	x
31	x	x	x	x

Probieren wir gleich einmal aus, wie ein Sprite mit 15 verschiedenen Farben aussieht:

```
REM MultiColor  Pfad: Darstellung/1ØAnimation/MultiColor
'P1Ø-6
DEFINT a-z
GOSUB LibraryOeffnen
GOSUB SpeicherReservieren :IF fehl THEN ende
PALETTE Ø,.6,.6,.6                         'grau
PALETTE 17,1,Ø,Ø     :PALETTE 18,Ø,1,Ø     'rot, gruen
PALETTE 19,Ø,Ø,1     :PALETTE 2Ø,.67,Ø,Ø   'blau, d.rot
PALETTE 21,Ø,.67,Ø   :PALETTE 22,Ø,Ø,.67   'd.gruen, d.blau
PALETTE 23,1,1,1     :PALETTE 24,Ø,Ø,Ø     'weiss, schwarz
PALETTE 25,.73,.6,.53:PALETTE 26,1,.73,.8  'braun, rosa
PALETTE 27,.8,Ø,.93  :PALETTE 28,1,.87,.73 'purpur, oker
PALETTE 29,.8,.8,.8  :PALETTE 3Ø,1,Ø,1     'grau, violett
PALETTE 31,1,1,Ø                           'gelb

FOR i = Ø TO 71 STEP 2
  READ sd%:POKEW d1&+i,sd% AND 65535&
NEXT
FOR i = Ø TO 71 STEP 2
```

```
  READ sd%:POKEW d2&+i,sd% AND 65535&
NEXT

POKEW s1&+4,16      'SpriteHoehe Sprite1
POKEW s2&+4,16      'SpriteHoehe Sprite2
num(1)=GetSprite&(s1&,2):IF num(1)=-1 THEN fehl=2:GOTO ende
num(2)=GetSprite&(s2&,3):IF num(2)=-1 THEN fehl=2:GOTO ende

'Bit 7 des 2.Datenwortes ATTACHED Wert 128
'fuer verknuepfte Sprites
'eingetragen in Sprite mit ungerader Nummer
POKEW d2&+2,PEEKW(d2&+2) OR 128

CALL ChangeSprite&(Ø,s1&,d1&)
CALL ChangeSprite&(Ø,s2&,d2&)

x1=Ø:y1=8Ø
CALL MoveSprite&(Ø,s1&,x1,y1)
CALL MoveSprite&(Ø,s2&,x1,y1)
LOCATE 5,5:PRINT "START = Taste"
WHILE INKEY-="":WEND
LOCATE 5,5:PRINT "ENDE = Taste "
b=-1:dx=1:dy=1
WHILE b
  x1=x1+dx:IF x1>WINDOW(2)/2 OR x1<Ø THEN dx=dx*(-1)
  y1=y1+dy:IF y1>WINDOW(3) OR y1<Ø THEN dy=dy*(-1)
  CALL MoveSprite&(Ø,s1&,x1,y1)
  CALL MoveSprite&(Ø,s2&,x1,y1)
  IF INKEY-<>"" THEN b=Ø
WEND

ende:
  FOR i=1 TO 2
    IF num(i)>Ø THEN CALL FreeSprite&(num(i))
  NEXT
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN
    ON fehl GOSUB F1,F2
    BEEP:PRINT ft-
  END IF
  LIBRARY CLOSE
END
```

```
F1:ft-= "Speicher nicht ausreichend":RETURN
F2:ft-= "kein freier Sprite":RETURN

LibraryOeffnen:
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION GetSprite&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
RETURN

SpeicherReservieren:
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  s1&=AllocRemember&(rk&,16,art&)   'fuer SpriteStruktur1
    IF s1&=Ø THEN fehl=1:RETURN
  s2&=AllocRemember&(rk&,16,art&)   'fuer SpriteStruktur2
    IF s2&=Ø THEN fehl=1:RETURN
  d1&=AllocRemember&(rk&,8Ø,art&)  'fuer SpriteDaten1
    IF d1&=Ø THEN fehl=1:RETURN
  d2&=AllocRemember&(rk&,8Ø,art&)  'fuer SpriteDaten2
    IF d2&=Ø THEN fehl=1
RETURN

BallEbeneØ_1:
DATA &hØØØØ,&hØØØØ
DATA &hØ3cØ,&hØØØØ, &hØØØØ,&hØffØ, &h3ffc,&h3ffc
DATA &hØØØØ,&hØØØØ, &h7ffe,&hØØØØ, &hØØØØ,&h7ffe
DATA &hffff,&hffff, &hØØØØ,&hØØØØ, &hØØØØ,&hØØØØ
DATA &hffff,&hØØØØ, &hØØØØ,&h7ffe, &h7ffe,&h7ffe
DATA &hØØØØ,&hØØØØ, &h3ffc,&hØØØØ, &hØØØØ,&hØffØ
DATA &hØ3cØ,&hØ3cØ
DATA &hØØØØ,&hØØØØ
BallEbene2_3:
DATA &hØØØØ,&hØØØØ
DATA &hØØØØ,&hØØØØ, &hØØØØ,&hØØØØ, &hØØØØ,&hØØØØ
DATA &h3ffc,&hØØØØ, &h7ffe,&hØØØØ, &h7ffe,&hØØØØ
DATA &hffff,&hØØØØ, &hØØØØ,&hffff, &hØØØØ,&hffff
DATA &hØØØØ,&hffff, &hØØØØ,&h7ffe, &hØØØØ,&h7ffe
DATA &h3ffc,&h3ffc, &h3ffc,&h3ffc, &hØffØ,&hØffØ
DATA &hØ3cØ,&hØ3cØ
DATA &hØØØØ,&hØØØØ
```

Na, habe ich zuviel versprochen?

10.4 Der Mauszeiger

Ein Sprite haben wir bisher außer acht gelassen. Dabei haben wir dieses Sprite fast immer vor unseren Augen. Richtig! Ich spreche von dem Sprite des Mauszeigers. Der Pointer, wie er auch genannt wird, ist das Hardware-Sprite Nummer 0. Natürlich wollen wir dieses spezielle Sprite erst recht nach Lust und Laune, oder besser gesagt nach den Erfordernissen verändern können.

Alles, was Sie über die Eigenschaften der Hardware-Sprites gelesen haben, gilt auch für den Pointer. So gelten für ihn die Farbnummern 16–19, die Breite darf maximal 16 Bildpunkte betragen etc. Der Befehlssatz der Hardware-Sprites kann logischerweise für den Pointer nicht gelten. Die Nummer eines Sprites brauchen Sie nicht zu holen, es ist immer der Sprite Null. Da die Bewegung des Pointer-Sprites mit der Maus erfolgt, ist auch eine Moove-Routine überflüssig. Verbleiben von den 4 SimpleSprite-Routinen noch 2. Die benötigt auch der Pointer. Da der Mauszeiger einige Daten mehr benötigt (zum Beispiel die Koordinaten für den heißen Punkt), ist der entsprechende Befehl etwas anders aufgebaut:

Format: `SetPointer(Window,Pointer,Height,Width,XOffset,YOffset)`

Der Parameter *Window* zeigt auf die Window-Struktur und *Height* und *Width* geben die Höhe und die Breite des Sprites an. Die beiden Werte *XOffset* und *YOffset* enthalten die Koordinaten des heißen Punktes, also des Punktes, der als Pointer-Position aus der Fenster-Struktur gelesen werden kann. Die Koordinaten erhalten die Entfernung in Bildpunkten von der linken unteren Ecke des Sprites. Bleibt noch der Parameter *Pointer*. Er zeigt auf die Pointer-Struktur. Wir benötigen also im Gegensatz zu den Hardware-Sprites nur eine Struktur. Diese Struktur entspricht der SimpleSprite-Struktur, mit dem Unterschied, daß am Schluß zusätzlich zwei Worte mit dem Wert Null angehängt werden.

Wenn Sie den normalen Pointer zurückholen, steht Ihnen eine Routine zur Verfügung, die als Parameter nur die Window-Struktur benötigt.

Format: `ClearPointer(Window)`

Es ist wirklich nicht viel, was zum Setzen eines neuen Mauszeigers und zum Zurückholen des Standard-Pointers benötigt wird. Wir verzichten deshalb auf ein Beispiel. Sie finden unter anderem im letzten Kapitel im Programm CAVE und in dem Zeichenprogramm die Programmierung von einfachen Mauszeigern. Nicht viel schwieriger gestaltet sich die Programmierung eines animierten Pointers. Unser Beispiel-Programm zeigt, wie einfach das ist:

```
REM Cutter  Pfad: Darstellung/1ØAnimation/Cutter
'P1Ø-7
CLEAR
DEFINT a-z
win&=WINDOW(7)         'Fenster-Struktur
rp&=PEEKL(win&+5Ø)     'RastPort-Struktur
dlg=56                 'Laenge Sprite-Struktur
GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
CALL InitDatas
PALETTE 17,.47,.47,.47
CALL SetRast&(rp&,3)
PRINT "Bitte Maus bewegen    Taste=ENDE"

a=-1 :altx=PEEKW(win&+14):alty=PEEKW(win&+12)
WHILE a
  x=PEEKW(win&+14):y=PEEKW(win&+12)
  IF x<>altx OR y<>alty THEN GOSUB AKTION
  altx=x:alty=y
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND

ende:
  CALL ClearPointer&(win&)
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN BEEP:PRINT "Speicher nicht ausreichend"
  LIBRARY CLOSE
END

LibraryOeffnen:
  DECLARE FUNCTION AllocRemember&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
RETURN

Speicher:
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  ms&=AllocRemember&(rk&,176,art&) 'fuer SpriteDaten
  IF ms&=Ø THEN fehl=2
RETURN
```

```
SUB InitDatas STATIC
SHARED ms&,dlg%
FOR d%= Ø TO 3*dlg%-1 STEP dlg%
  FOR i% = Ø TO dlg%-1 STEP 2
    READ sd%:POKEW ms&+d%+i%,sd% AND 65535&
NEXT i%,d%
END SUB

AKTION:
CALL SetPointer&(win&,ms&+dif,11,16,-6, 6)
dif=dif+dlg:IF dif >111 OR dif<55 THEN dlg=dlg*(-1)
LINE (x,y)-(altx,alty),Ø
RETURN

SchereAuf:
DATA &hØ,&hØ
DATA &h8ØØØ,&hØØØf, &heØØØ,&hØØØ9, &h38ØØ,&hØØ1f
DATA &h1eØØ,&hØØ6Ø, &hØ7ØØ,&hØØcØ, &hØ1ØØ,&hØ38Ø
DATA &hØØØØ,&hØ78Ø, &hØØ6Ø,&hØeØØ, &hØØ1f,&h38ØØ
DATA &hØØØ9,&h6ØØØ, &hØØØf,&h8ØØØ
DATA &HØ,&HØ,&HØ,&HØ
SchereMitte:
DATA &hØ,&hØ
DATA &hØØØØ,&hØØØØ, &hØØØØ,&hØØØf, &h8ØØØ,&hØØØ9
DATA &hfØØØ,&hØØ3f, &h3cØØ,&hØ1cØ, &hØ9ØØ,&hØ78Ø
DATA &hØØcØ,&hØfØØ, &hØØ3f,&h38ØØ, &hØØØ9,&hcØØØ
DATA &hØØØf,&hØØØØ, &hØØØØ,&hØØØØ
DATA &hØØØØ,&hØØØØ, &hØØØØ,&hØØØØ
SchereZu:
DATA &hØØØØ,&hØØØØ
DATA &hØØØØ,&hØØØØ, &hØØØØ,&hØØØØ, &hØØØØ,&hØØØf
DATA &heØØØ,&hØØØ9, &hfcØØ,&hØ1ff, &hØ1ØØ,&h1fØØ
DATA &hØØff,&heØØØ, &hØØØ9,&heØØØ, &hØØØf,&hØØØØ
DATA &hØØØØ,&hØØØØ, &hØØØØ,&hØØØØ
DATA &hØØØØ,&hØØØØ, &hØØØØ,&hØØØØ
```

Zuerst öffnen wir die Intuition-Library und, da auch ein Grafik-Befehl eingesetzt wird, auch die Grafik-Bibliothek. Anschließend reservieren wir den Speicherplatz für drei Pointer-Strukturen. Im Unterprogramm »InitDatas« werden die Daten gelesen und in den reservierten Speicher geschrieben. Nun wird das Fenster mit *SetRast* in der orangenen Farbe 3 gefüllt.

Das Hauptprogramm läuft wieder einmal in einer WHILE/WEND-Schleife ab. Aus den Speicherstellen 12 und 14 der Window-Struktur werden die aktuellen Maus-Koordinaten geholt. Diese werden mit den Koordinaten des vorhergegangenen Schleifendurchganges verglichen. Wenn die Position verändert wurde, hat der Anwender die Maus bewegt, oder er bewegt sie noch. In diesem Fall wird zur Subroutine »AKTION« verzweigt.

Bei »AKTION« wird zuerst mit *SetPointer* ein neuer Mauszeiger gesetzt. Der Parameter *Pointer* zeigt auf die Pointer-Struktur. Im Programm sind nun die Pointer-Strukturen hintereinandergelegt. Den Abstand zur nächsten Struktur enthält die Variable *dif.* Bei jedem neuen Aufruf der Subroutine wird die Distanz *dif* nach oben erhöht und dann wieder nach unten vermindert. Dadurch werden nacheinander die Sprites 1,2,3,2,1,2 etc. ausgegeben. Die drei Sprites stellen eine Schere mit drei verschiedenen Öffnungswinkeln dar. Im Klartext bedeutet das, solange der Anwender die Maus bewegt, geht die Schere auf und zu. Um den Eindruck des Schneidens zu verstärken, wird eine Linie zur letzten Position in der Hintergrundfarbe gezogen. Die Schere schneidet sich sozusagen einen Weg durch das orangefarbene Papier. Die Werte für *XOffset* und *YOffset* wurden dabei so gewählt, daß die Schnittlinie in der Mitte der Schere beginnt.

Durch den Druck auf eine beliebige Taste wird das Programm beendet. Mit *ClearPointer* wird der normale Mauszeiger wieder hervorgezaubert. Nun muß nur noch der belegte Speicher freigegeben und die Library geschlossen werden.

Nachdem Sie wissen, wie einfach die Angelegenheit ist, werden Sie zukünftig in Ihren Programmen sicherlich auch eigene Pointer einsetzen.

10.4.1 Mauszeiger abschalten

Die Maus mit ihrem zugehörigen Mauszeiger ist eine feine Sache. Abgesehen von einer kurzen Eingewöhnungszeit werden Sie darauf inzwischen sicherlich nicht mehr verzichten können. Manchmal allerdings stört der Pfeil gewaltig. Die beste Lösung wäre es, könnte man den Pointer für eine gewisse Zeit aus- und dann wieder einschalten.

Mit den Kenntnissen, die Sie inzwischen über die Pointer-Programmierung errungen haben, sollte das keine besonders große Herausforderung sein. Natürlich! Das ist die Lösung! Wir konstruieren einfach einen neuen Mauszeiger, der in Wirklichkeit gar keiner ist. Anders ausgedrückt, setzen wir einen Pointer mit der Höhe Null und mit einem leeren Datenfeld.

```
REM PointerLess Pfad: Darstellung/1ØAnimation/PointerLess
'P1Ø-8
DECLARE FUNCTION AllocRemember&() LIBRARY
LIBRARY ":bue/intuition.library"
```

```
art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
ms&=AllocRemember&(rk&,16,art&)
IF ms&=Ø THEN fehl=1 :GOTO ende

CALL SetPointer&(WINDOW(7),ms&,Ø,Ø,Ø,Ø)

a=-1
WHILE a
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND

ende:
  CALL ClearPointer&(WINDOW(7))
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN BEEP:PRINT "Speicher nicht ausreichend"
  LIBRARY CLOSE
END
```

Drei Zeilen des Programmes sind zum Löschen und Zurückholen des Pointers erforderlich. Mit *AllocRemember* wird der Speicher für die Pointer-Struktur reserviert. Mit dieser Adresse wird nun der Pointer mit einer Höhe von Null gesetzt. Ein Tastendruck holt durch die Routine *ClearPointer* den Standard-Mauszeiger zurück. Alles andere ist notwendiges Beiwerk, das wahrscheinlich für andere Library-Routinen in Ihrem Programm sowieso vorhanden ist.

Sie sehen, der Mauszeiger verschwindet spurlos und erscheint am Ende des Programmes wieder, so als sei nichts gewesen. Probieren Sie einmal, während der Pointer abgeschaltet ist, einen Menüpunkt auszuwählen. Sie werden feststellen, der Pointer ist immer noch aktiv, nur zu sehen ist er nicht.

Grafik mit AMIGA-BASIC

View-Modi

3. Teil

Commodore Sachbuch

Kapitel 11

Riesengrafik

Im Kapitel über die Screens und Windows am Anfang des Buches habe ich versprochen, Ihnen zu zeigen, wie die interessanten View-Modi des Amiga programmiert werden können. Dieses Versprechen will ich nun einlösen. In diesem Teil des Buches werden wir die speziellen View-Modi des Amiga genau unter die Lupe nehmen. Die folgende Tabelle faßt die einzelnen Modi mit den Werten der einzelnen Flags, die wir für die Programmierung benötigen, zusammen.

View-Modus	**hex (Bit)**	**dezimal**	**Kurzbeschreibung**
LORES	$0 (–)	0	320 Pixel horizontal
GENLOCK__VIDEO	$2 (1)	2	für Genlock Video Interface
LACE	$4 (2)	4	Zeilensprungverfahren (Höhe*2)
PFBA	$40 (6)	64	PlayField-Priorität für DUALPF
EXTRAHALFBRITE	$80 (7)	128	Modus für 64 Farben
DUALPF	$400 (10)	1024	zwei PlayFields
HAM	$800 (11)	2048	Hold and Modify (4096 Farben)
SPRITES	$4000 (14)	16384	für Hardware-Sprites
HIRES	$8000 (15)	32768	640 Pixel horizontal

Wenn Sie sich die Tabelle genauer ansehen, werden Sie feststellen, daß die Überschrift des Kapitels mit den View-Modi fast nichts gemeinsam hat. Andererseits hat zum Beispiel der Modus DUALPF wenig Sinn, wenn Sie nicht vorher über Super-Bitmaps und PlayField-Scrolling Bescheid wissen.

11.1 Von der Hardware zum Video-Bild

Im Zusammenhang mit der Darstellung des Video-Bildes haben Sie inzwischen eine Menge Begriffe kennengelernt. Von Screens und Windows war da die Rede, ViewPort und RastPort werden bei den Routinen der Grafik-Library benötigt, ColorMap und

ColorTable sind irgendwie für die Farben verantwortlich, und alles scheint irgendwie mit der Bitmap zusammenzuhängen. Um Grafiken zwischenzuspeichern, haben wir schon eigene Bitmaps programmiert, ein andermal verwendeten wir dafür zusätzliche Screens. Ganz schön verwirrend das Ganze, finden Sie sicherlich. Nun, wie bereits einmal erwähnt, habe ich diese Thematik bereits in meinem Buch »Programmierpraxis Amiga-Basic« ausführlich besprochen. Ich will nun versuchen, in Kurzform den Zusammenhang zu erklären. Wie so oft, so wird auch hier eine kleine Grafik mehr sagen als viele Worte.

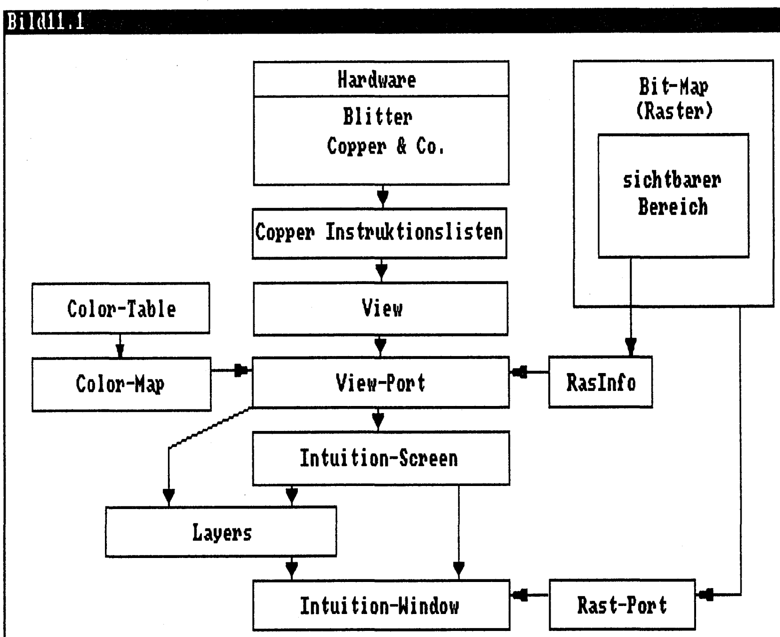


Bild 11.1: Übersicht über die Grafik-Ausgabe-Elemente

Die Skizze enthält die wichtigsten Elemente, die zum Verständnis der Zusammenhänge notwendig sind. Das Grund-Element bildet der *View*. Er ist die Schnittstelle zur Hardware des Amiga.

Der *ViewPort* ist sozusagen fast schon ein Screen. In ihm sind alle wichtigen Informationen für die Bildausgabe enthalten. Dazu zählen die Farbinformationen, die Lage des Bildes auf dem Bildschirm, die Bildbreite und -höhe, und natürlich der View-Modus. Über die RasInfo-Struktur ist die Verbindung zur Bitmap gegeben. Sie können auch mehrere ViewPorts gleichzeitig auf dem Bildschirm darstellen. Sie dürfen sich jedoch nicht überlappen und müssen durch eine Bildzeile voneinander getrennt sein. Zum ein-

facheren Verständnis und wegen der besseren Übersicht habe ich in der grafischen Zusammenstellung darauf verzichtet.

Die Intuition verwaltet in einem Screen die ganzen Strukturen, die zum Bildaufbau notwendig sind. So sind die Strukturen des ViewPorts, des RastPorts, der Bitmap und für die LayerInfo direkter Bestandteil eines Screens. Über den Rastport, der alle Bestandteile enthält, die für die zeichnerische Darstellung notwendig sind, besteht eine weitere Verbindung zur Bitmap. Die direkte Verbindung zum ersten Fenster des Screens darf selbstverständlich auch nicht fehlen.

Ein Layer stellt sozusagen den Rohling für ein Fenster dar. Aus diesem Layer macht dann die Intuition ein Window mit den Borders, der Titelleiste und den System-Gadgets. Eine direkte Bildausgabe über die Layers ist auch ohne Window möglich. In der Regel arbeiten wir jedoch mit einem oder mehreren Windows, die ein Screen haben können.

Die zweite Verbindung zur Hardware stellt die Bitmap dar. Eine Bitmap ist eigentlich nichts weiter als ein gewisser Bereich des RAM-Speichers. Die Software behandelt diesen Speicherabschnitt als rechteckigen Bereich, der aus einer oder bis zu sechs Ebenen übereinander besteht und somit für die Farben oder einen Spezial-Modus verantwortlich zeichnet. Daß jedes Bit dieser Speicher-Karte einer Farbinformation eines Grafik-Punktes entspricht, haben wir bereits besprochen.

Ich bin mir sicher, daß diese kurze Zusammenfassung zum Verständnis der Zusammenhänge beigetragen hat. Weitere Informationen finden Sie in den einzelnen Strukturen, im Anhang dieses Buches.

11.2 Ausschnitt aus einem Raster

Wie Sie inzwischen wissen, gibt es für jeden Bildpunkt auf Ihrem Bildschirm eine oder mehrere (Farbe) Speicherstellen in einer Bitmap. Das bedeutet, daß die Bitmap genauso groß sein muß wie der Screen des sichtbaren Fensters. Wenn ich genauso groß sage, meine ich, daß sie nicht kleiner sein darf, größer müßte eigentlich möglich sein. Als Anwender-Bitmap ist es auch möglich. Der Amiga gestattet eine Bitmap-Größe von 1024 mal 1024 Pixel.

Die Erstellung einer eigenen Bitmap haben wir bereits praktiziert. Mit dem gleichen Verfahren läßt sich auch eine Riesen-Bitmap programmieren. Fassen wir die einzelnen Schritte kurz zusammen:

- Reservieren eines Speicherbereiches für die Bitmap-Struktur, zum Beispiel mit *AllocRemember.*
- Reservieren bzw. Belegen einer, oder je nach Tiefe mehrerer Bit-Ebenen mit *AllocRaster.*
- Löschen des eventuell vorhandenen Daten-Schrotts der reservierten BitPlanes mit *BltClear.*
- Eintragen der Daten in die neue Bitmap-Struktur durch POKEs.
- Initialisieren der neuen Bitmap mit *InitBitMap.*

Damit haben wir nun die neue Bitmap. Als nächstes müssen wir die Bitmap in das System einbinden. Das geht am einfachsten mit einer NewScreen-Struktur. Im Feld *Type* dieser Struktur werden die Bits für CUSTOMSCREEN ($F) + CUSTOMBITMAP ($40) gesetzt. Die Adresse der Bitmap wird dann noch in das Feld *CustomBitMap* eingetragen. Damit wären wir bereits fertig.

Ein kleines Problem bleibt noch. Wir haben den oberen linken Bildausschnitt aus der Riesen-Bitmap auf dem Bildschirm. Wie können wir einen anderen Bereich der Bitmap sichtbar machen? Schauen wir uns dazu die RasInfo-Struktur an. Dort finden wir in Wortlänge die beiden Felder für *RxOffset* (ri+8) und *RyOffset* (ri+10). Diese beiden Variablen stellen den Abstand des darzustellenden Bitmap-Ausschnittes zur oberen linken Ecke (0,0) dar. *RxOffset* ist der Abstand in Pixel zwischen linker Bitmap-Kante und linker Kante des Bildausschnittes. Analog ist *RyOffset* die Distanz der beiden oberen Kanten.

Wenn wir einfach nur die neuen Werte in die Speicherstellen poken, wäre das ein gutes Mittel, einen Gruß vom mächtigen GURU zu erhalten. Führen Sie sich das Bild der Ausgabe-Elemente noch einmal zu Gemüte, dann wissen Sie auch warum. Die einzelnen Grafik-Elemente müssen nämlich neu aufeinander abgestimmt werden. Wir sagen der Intuition einfach, daß sie die ganze Ausgabe neu überdenken soll. Die Routine dazu trägt den gleichen Namen: *RethinkDisplay.* Zum Schluß bringen wir dann mit *ScrollVPort* den ViewPort, bzw. den Screen auf die neue Position in der Super-Bitmap. Im Zusammenhang sieht die Prozedur folgendermaßen aus:

```
POKEW ri&+8,x
POKEW ri&+1Ø,y
CALL RethinkDisplay&
CALL ScrollVPort&(vp&)
```

Ein Problem sollten Sie bei der Auswahl der Größe der Anwender-Bitmap nicht aus den Augen verlieren. Ich spreche vom benötigten Speicherbereich, der in den unteren 512 Kbyte des Amiga Platz finden muß. Eine Bitmap von 1024 mal 1024 und einer Tiefe von 2 verschlingt satte 262144 Byte!

11.3 Eine Kirche auf 1024 mal 1024 Grafikpunkten

Wenn schon die Möglichkeit einer Grafik von 1024 * 1024 Bildpunkten besteht, dann wollen wir sie auch nutzen. Um den Speicher nicht zu sehr zu strapazieren, begnügen wir uns jedoch mit zwei Farben, also einer Farbtiefe von 1. Das Programm schiebt den Bildschirm reihenweise, im Abstand von 50 Bildpunkten in X-Richtung und von 128 Pixel in Y-Richtung, über die Bitmap. Der Screen ist, um die Größe der Super-Bitmap zu unterstreichen, von niedriger Auflösung.

Haben Sie schon nachgerechnet, wie oft ein Screen im View-Modus LORES in diese Bitmap hineinpaßt? Richtig! Die Bitmap ist 12,8 mal so groß wie ein Bildschirm, der 256 Grafik-Reihen hoch ist. Bei einer Bildschirmhöhe von 200 Bildpunkten würden erst 16,4 Screens die Super-Bitmap füllen.

```
REM BigPlane  Pfad: Modus/11Riesengrafik/BigPlane
'P11-1
CLEAR
DEFINT a-z :DIM ki(27),f1(35),fa(35),f2(29)
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
sb=320:tiefe=1:vm=0:tp=79:tp1=15:stit&=0
wb=sb:wh=sh:idf&=8:fl&=4096
t2-="BIG PLANE    Bitte etwas Geduld"+CHR-(0):wt2&=SADD(t2-)
t1-="BIG PLANE    linke Maustaste=ENDE"+CHR-(0):wt1&=SADD(t1-)
GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
GOSUB BManlegen :IF fehl THEN ende
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,mb&) :IF fehl THEN ende
WINDOW CLOSE 1
CALL Fenster (wb,wh,idf&,fl&,wt2&,tp1) :IF fehl THEN ende
GOSUB zeichnen
CALL SetWindowTitles&(win&,wt1&,-1)
start:
FOR y=0 TO 768 STEP 128
  FOR x=0 TO 704 STEP 50
    POKEW ri&+8,x:POKEW ri&+10,y
    CALL RethinkDisplay&
    CALL ScrollVPort&(vp&)
    tt&=TIMER:WHILE TIMER<tt&+1:WEND
NEXT x,y
POKEW ri&+8,0:POKEW ri&+10,0
CALL RethinkDisplay&
CALL ScrollVPort&(vp&)
```

```
a=-1
WHILE a
  me&=PEEKL(win&+86)
  abfrage: mess&=GetMsg&(me&):IF mess&=Ø THEN abfrage
  cla&=PEEKL(mess&+2Ø)     'Class of IntuitionMessage
  cod=PEEKW(mess&+24)      'Code of IntuitionMessage
  CALL ReplyMsg&(mess&)
  IF cla&=8 THEN a=Ø
WEND

ende:
IF win& THEN CALL CloseWindow&(win&)
IF scr& THEN CALL CloseScreen&(scr&)
FOR i = Ø TO tiefe-1
  IF be&(i) THEN CALL FreeRaster&(be&(i),bmbr,bmRows)
NEXT
IF rk& THEN CALL FreeRemember&(rk&,-1)
IF fehl THEN
  ON fehl GOSUB f1,f2,F3
  BEEP:PRINT ft-
END IF
LIBRARY CLOSE :ERASE ki,f1,fa,f2
SYSTEM

f1:ft-= "Fehler bei OpenWindow":RETURN
f2:ft-= "Speicher nicht ausreichend":RETURN
F3:ft-= "FEHLER bei OpenScreen":RETURN

LibraryOeffnen:
DECLARE FUNCTION AllocRaster&() LIBRARY
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION OpenScreen&() LIBRARY
DECLARE FUNCTION OpenWindow&() LIBRARY
DECLARE FUNCTION GetMsg&() LIBRARY
LIBRARY ":bue/intuition.library"
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/exec.library"
RETURN

Speicher:
art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
ms&=AllocRemember&(rk&,4Ø,art&) 'fuer NScreen-Struktur
IF ms&=Ø THEN fehl=2 :RETURN
```

```
mw&=AllocRemember&(rk&,48,art&) 'fuer NWindow-Struktur
IF mw&=Ø THEN fehl=2 :RETURN
mb&=AllocRemember&(rk&,1ØØ,art&)'fuer BitMap-Struktur
IF mb&=Ø THEN fehl=2 :RETURN
RETURN

zeichnen:
CALL SetRGB4&(vp&,Ø,Ø,Ø,Ø)
CALL SetRGB4&(vp&,1,15,Ø,Ø)
'Kirche
FOR i=Ø TO 27:READ ki(i):NEXT
CALL Move&(rp&,1ØØ,1Ø2Ø)
CALL PolyDraw&(rp&,14,VARPTR(ki(Ø)))
'Fenster1
FOR i=Ø TO 35:READ f1(i):NEXT
FOR x= 15Ø TO 9ØØ STEP 1ØØ
  FOR i=Ø TO 35 STEP 2
    fa(i)=f1(i)+x:fa(i+1)=f1(i+1)+81Ø
  NEXT
  CALL Move&(rp&,x,81Ø)
  CALL PolyDraw&(rp&,18,VARPTR(fa(Ø)))
NEXT x
FOR y= 35Ø TO 85Ø STEP 15Ø
  FOR i=Ø TO 35 STEP 2
    fa(i)=f1(i)+3Ø:fa(i+1)=f1(i+1)+y
  NEXT
  CALL Move&(rp&,3Ø,y)
  CALL PolyDraw&(rp&,18,VARPTR(fa(Ø)))
NEXT y
'Fenster2
FOR i=Ø TO 29:READ f2(i):NEXT
FOR x= 3ØØ TO 75Ø STEP 5Ø
  FOR i=Ø TO 29 STEP 2
    fa(i)=f2(i)+x:fa(i+1)=f2(i+1)+6ØØ
  NEXT
  CALL Move&(rp&,x,6ØØ)
  CALL PolyDraw&(rp&,15,VARPTR(fa(Ø)))
NEXT x
FOR x= 15Ø TO 9ØØ STEP 5Ø
  FOR i=Ø TO 29 STEP 2
    fa(i)=f2(i)+x:fa(i+1)=f2(i+1)+7ØØ
```

```
  NEXT
  CALL Move&(rp&,x,7ØØ)
  CALL PolyDraw&(rp&,15,VARPTR(fa(Ø)))
NEXT x
'Wolken
FOR wo=Ø TO 2Ø
  x=RND*8ØØ+15Ø: y=RND*4ØØ+3Ø
  FOR me= Ø TO 3Ø
    xx=RND*2Ø+x:yy=RND*1Ø+y
    xr=RND*5Ø:yr=RND*3Ø
    CALL DrawEllipse&(rp&,xx,yy,xr,yr)
NEXT me,wo
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
SHARED ms&,scr&,rp&,vp&,fehl%,ri&
StrukturNeuerScreen:
POKEW ms&,Ø      :POKEW ms&+2,Ø     'linke u.obere Kante
POKEW ms&+4,sb% :POKEW ms&+6,sh%    'Breite, Hoehe
POKEW ms&+8,tie%                    'Tiefe
POKE ms&+1Ø,Ø    :POKE ms&+11,1     'DetailPen, BlockPen
POKEW ms&+12,vm%                    'ViewModes
POKEW ms&+14,tp%                    'Type  CUSTOMSCREEN CUSTOMBITMAP
POKEL ms&+16,Ø                      'TextAttr
POKEL ms&+2Ø,stit&                  'Screen-Titel
POKEL ms&+24,Ø  :POKEL ms&+28,sbm& 'Gadget,BitMap
scr&= OpenScreen&(ms&)
IF scr&=Ø THEN fehl%=3
rp&=scr&+84 :vp&=scr&+44:ri&=PEEKL(scr&+8Ø)
END SUB

SUB Fenster (wb%,wh%,idf&,fl&,wtit&,tp%) STATIC
SHARED mw&,scr&,win&,fehl%
StrukturNeuesFenster:
POKEW mw&,Ø      :POKEW mw&+2,Ø    'linke u.obere Kante
POKEW mw&+4,wb% :POKEW mw&+6,wh%   'Breite, Hoehe
POKE mw&+8,1     :POKE mw&+9,Ø     'DetailPen, BlockPen
POKEL mw&+1Ø,idf&                  'IDCMPFlag MOUSEBUTTONS
POKEL mw&+14,fl&                   'Flags ACTIVATE
POKEL mw&+26,wtit&                 'Window-Titel
POKEL mw&+3Ø,scr&                  'Screen
POKEW mw&+38,1ØØ:POKEW mw&+4Ø,3Ø  'min. Breite und Hoehe
```

```
POKEW mw&+42,wb%:POKEW mw&+44,wh% 'max. Breite und Hoehe
POKEW mw&+46,tp%                  'Type  CUSTOMSCREEN
win&= OpenWindow&(mw&)
IF win&=Ø THEN fehl%=1
END SUB

BManlegen:
bmbr=1Ø24
bmBytPerRow=bmbr/8      'Bytes per Grafikzeile
bmRows=1Ø24              'Grafik-Zeilen
volum&=bmBytPerRow*bmRows
FOR i = Ø TO tiefe-1
  be&(i)=AllocRaster&(bmbr,bmRows)
  IF be&(i)=Ø THEN fehl=2:RETURN
  CALL BltClear&(be&(i),volum&,Ø)
NEXT
POKEW mb&,bmBytPerRow          'Bytes/Reihe
POKEW mb&+2,bmRows             'Reihen
POKE mb&+5,tiefe               'Tiefe
FOR n = Ø TO tiefe-1
  POKEL mb&+8+(n*4),be&(n)     'n BitPlanes
NEXT
CALL InitBitMap&(mb&,tiefe,bmbr,bmRows)
RETURN

Kirche:
DATA 1ØØ,1Ø2Ø,1Ø,1Ø2Ø,1Ø,3ØØ,1ØØ,3ØØ,45,1Ø,1Ø,3ØØ
DATA 1ØØ,3ØØ,1ØØ,1Ø2Ø,1Ø1Ø,1Ø2Ø,1Ø1Ø,75Ø,1ØØ,75Ø
DATA 35Ø,5ØØ,67Ø,5ØØ,1Ø1Ø,75Ø

fen1:
DATA Ø,Ø,Ø,125,5Ø,125,5Ø,Ø,25,Ø,25,125,Ø,125,5Ø,125
DATA 5Ø,1ØØ,Ø,1ØØ,Ø,75,5Ø,75,5Ø,5Ø,Ø,5Ø,Ø,25,5Ø,25,5Ø,Ø,Ø,Ø

fen2:
DATA Ø,Ø,1Ø,Ø,15,1Ø,-5,1Ø,-5,2Ø,15,2Ø,15,3Ø,-5,3Ø
DATA -5,1Ø,Ø,Ø,-5,1Ø,5,1Ø,5,3Ø,15,3Ø,15,1Ø
```

Nach der Dimensionierung der Variablen-Felder werden die Variablen für den neuen Bildschirm und das neue Fenster bereitgestellt. Für die Zeichen-Routinen könnten wir auf das Fenster verzichten. Wegen der notwendigen Kommunikation über den IDCMP benötigen wir es aber doch. In der Subroutine »LibraryOeffnen« werden die Funktionen für die Speicherreservierung, für die Open-Routinen und zum Abruf der Messages deklariert. Dazu benötigen wir die Intuition-, die Grafik- und die Exec-Library.

Die zweite Subroutine »Speicher« zeichnet für die notwendige Speicherreservierung verantwortlich. Nun legen wir bei »Bitmapanlegen« die Super-Bitmap an. Die Details dazu haben wir bereits besprochen. Im Unterprogramm »Schirm« wird die NewScreen-Struktur erstellt und der neue Bildschirm geöffnet. Von Interesse sind dabei, wie ebenfalls bereits erwähnt, das Feld für die Screen-Type (CUSTOMSCREEN+ CUSTOMBITMAP) und das Feld mit der Bitmap-Adresse.

Das nächste Unterprogramm »Fenster« enthält die NewWindow-Struktur und seine Aktivierung. Wichtig für die Abfrage des IDCMP ist das Flag MOUSEBUTTONS. Erwähnenswert ist noch das Feld für den Fenster-Typ, das mit dem Wert für CUSTOMSCREEN versorgt ist. Um Speicher zu sparen, schließen wir vorher das Basic-Fenster.

Kommen wir nun zur Subroutine »zeichnen«. Zuerst setzen wir die Hintergrundfarbe auf Schwarz und die Vordergrundfarbe auf Rot, um einen guten Kontrast zu erhalten. Die meisten Eckwerte der Grafik sind in DATAs gespeichert. Sie werden eingelesen und an die Variablen-Felder übergeben. Die Zeichnung selbst erfolgt mit *Move* und *Poly-Draw*. Nacheinander werden so der Umriß der Kirche, 8 Kirchenfenster, 4 Kirchturmfenster und 26 Giebelfenster gezeichnet. 21 Wolken füllen den Himmel über dem Kirchendach.

Mit *SetWindowTitles* wird eine neue Information an den Anwender ausgegeben. Beim Label »start« folgt nun die Verschiebung des Bildausschnittes der Bitmap wie vorher beschrieben. Zum Schluß wird wieder das ursprüngliche Bild gezeigt. In der WHILE/ WEND- Schleife wartet das Programm, bis der Anwender die linke Maustaste betätigt. Die Schleife wird verlassen, und bei »ende« beginnen die Aufräumungsarbeiten. Es werden das Fenster und der Screen geschlossen, die Speicherbereiche der Bitmap und der Strukturen freigegeben und die Libraries geschlossen.

11.4 PlayField-Scrolling

Auch das Standard-Basic des Amiga kennt eine SCROLL-Anweisung. Leider ist diese auf den Ausschnitt eines Fensters beschränkt und kann daher zum Rollen des kompletten Bildschirms oder gar einer größeren Bitmap nicht eingesetzt werden. Für viele Anwendungsbereiche können und wollen wir jedoch auf ein echtes Playfield-Scrolling nicht verzichten. Wie sonst sollten Sie mit einem Flugzeug über eine schier endlose Fläche gleiten oder mit einem Boot eine scheinbar unbegrenzte Wasserwüste durchqueren können? Wie wollen Sie sonst eine technische Zeichnung in einer Super-Bitmap bearbeiten?

Die Angelegenheit ist eigentlich gar nicht so kompliziert zu lösen. Genaugenommen kennen Sie den einen Weg zum Playfield-Scrolling bereits. Ich sagte den einen Weg,

da es auch hier, wie so oft im Leben, verschiedene Wege gibt, die ans Ziel führen. Sie werden nun zwei dieser Wege kennenlernen. Welchen der beiden Sie in Ihren Programmen einsetzen werden, wird letztlich von den Erfolgen oder Mißerfolgen abhängen, die Sie mit den beiden Möglichkeiten haben.

11.4.1 Der rollende ViewPort

Sie haben richtig gelesen. Wir rollen einfach den ViewPort oder den Screen über die Bitmap. Die Technik ist die gleiche, wie wir sie beim Betrachten des Bildes im Programm *BigPlane* angewendet haben. Der einzige Unterschied besteht darin, daß wir zum Scrollen die Verschiebung Pixel für Pixel vornehmen. Dabei tritt ein interessanter Effekt zu Tage. Aber schauen wir uns das Prinzip erst in einem einfachen Beispiel an.

```
REM Ringelspiel  Pfad: Modus/11Riesengrafik/Ringelspiel
'P11-2
CLEAR
DEFINT a-z
LIBRARY ":bue/intuition.library"
LIBRARY ":bue/graphics.library"
b=WINDOW(2):h=WINDOW(3):win&=WINDOW(7):scr&=PEEKL(win&+46)
vp&=scr&+44:ri&=PEEKL(vp&+36):PALETTE 2,Ø,1,Ø
FOR i=Ø TO 1ØØ
 CIRCLE (RND*b,RND*h),RND*h/2,RND*3:NEXT
PRINT "ENDE = beliebige Taste"
a=-1
WHILE a
    POKEW ri&+8,Dx
    CALL RethinkDisplay&
    CALL ScrollVPort&(vp&)
    Dx=Dx+1:IF Dx>639 THEN Dx=Ø
    ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
  POKEW ri&+8,Ø
  CALL RethinkDisplay&
  CALL ScrollVPort&(vp&)
LIBRARY CLOSE :END
```

Sie sehen, das Programm ist wirklich ganz einfach. In der WHILE/WEND-Schleife wird der Wert für *Dx* jeweils um 1 erhöht, bis die Bitmap-Breite von 640 erreicht ist. Dann beginnt der Scroll-Vorgang von vorne. Sobald eine beliebige Taste gedrückt wird, wird die Schleife verlassen und *Dx* auf Null, also auf die Standard-Position gesetzt.

Wie Sie an dem Beispiel sehen konnten, wird das Bild tatsächlich um den ViewPort gescrollt. Der Bildabschnitt, der links vom Monitor verschwindet, kommt auf der rechten Seite wieder hervor. Das funktioniert auch, wenn Sie die Werte für *Dx* größer als 640 werden lassen. Probieren Sie es ruhig einmal aus, der Scroll-Vorgang wird trotzdem präzise abgewickelt. Damit haben wir ein echtes, kreisförmiges Scrollen realisiert. Ganz anders sieht die Sache beim Scrollen um die Y-Achse aus. Ersetzen Sie dazu in dem Beispiel die Variable *Dx* durch *Dy* und die beiden Zeilen

```
POKEW ri&+8,Dx     (und POKEW ri&+8,Ø)
durch
POKEW ri&+1Ø,Dy    (und POKEW ri&+1Ø,Ø).
```

Sie werden feststellen, daß das Bild (bzw. die Bitmap) nach oben aus dem Monitor wandert, ohne daß es von unten wieder hervorkommt. Sie sehen statt dessen die Überbleibsel von Daten aus anderen Speicherbereichen. Wenn Sie also eine Super-Bitmap in Y-Richtung rollen wollen, müssen Sie im Programm die Grenzen der Bitmap abfragen.

11.4.2 Ein rollendes Layer

Die zweite Variante des Playfield-Scrolling ist nicht so komfortabel. Normalerweise zeichnen wir unsere Grafiken in ein Window, oder wenn wir es nicht benötigen, direkt in den Screen. Da wir ein Fenster nicht rollen können, müssen wir uns eine Stufe nach unten begeben. Ich spreche von den Layers, die wir einmal bereits kurz angesprochen haben. Die einzelnen Routinen für die Layers sind in der Layers-Library abgelegt. In dieser Library finden wir auch die Routine *ScrollLayer*, die, wie schon der Name sagt, ein Scrolling ermöglicht.

Bevor wir die Routine einsetzen können, benötigen wir noch ein paar Hintergrundinformationen über die Layers. Da wir keine Bildausgabe mit den Routinen der Grafik-Library erstellen wollen, was auch vom Basic aus durchaus möglich ist, müssen wir die Layer-Parameter mit denen des Fensters in Einklang bringen. Schauen wir uns unter diesem Gesichtspunkt die Routine zum Erzeugen eines UpFront-Layers an. Ein UpFront-Layer liegt vor allen anderen Layers.

Format: `Layer=CreateUpFrontLayer(li,bm,xØ,yØ,x1,y1,flags,±bm2[)`

Das Ergebnis, die Variable *Layer*, erhält die Adresse der neu geschaffenen Layer-Struktur zugewiesen. Konnte das neue Layer nicht erzeugt werden, so wird Null zurückgegeben. Die komplette Layer-Struktur finden Sie wie üblich im Anhang. Nun zu den einzelnen Parametern.

li ist ein Zeiger auf eine LayerInfo-Struktur. Sie finden die zugehörige Adresse an der 224. Speicherstelle der Screen-Struktur.

Der Parameter *bm* zeigt auf die Bitmap, die von allen anderen Layers benutzt wird. Damit ist die normale Bitmap der Video-Ausgabe gemeint. Sie werden gleich verstehen, warum extra darauf hingewiesen wird.

x0 und *y0* markieren die linke obere Ecke und *x1* und *y1* die rechte untere Ecke des Layers. Diese Parameter sind für uns deshalb interessant, weil wir sie aus der Layer-Struktur des bestehenden Fensters holen und für das neue Layer einsetzen werden. Damit können wir sicherstellen, daß das existierende Fenster und das darübergelegte Layer gleich groß werden.

Die *flags* sagen aus, welche Layer-Type erstellt werden soll. Wir benötigen das Bit 2 (Wert 4) für ein Layer mit Super-Bitmap.

Bleibt noch der letzte Parameter *bm2*. Wenn bei den Flags das 2. Bit für ein Layer mit eigener Riesen-Bitmap gesetzt ist, wird hier die Adresse der zugehörigen Bitmap-Struktur eingetragen. Die Größe dieser Bitmap darf, genau wie bei der Standard-Bitmap, 1024 mal 1024 Bildpunkte betragen. Diese Layer-Bitmap hat mit der normalen Bitmap überhaupt nichts zu tun. Sie wird also separat verwaltet.

Mit diesem Wissen ausgerüstet können wir bereits das neue Layer erstellen. Allerdings ist noch ein kleiner Haken dabei. Das neue Layer verhält sich fast genauso wie ein Window. Wenn Sie also mit dem Mauspfeil darauf zeigen und klicken, wird es aktiv. Damit ist die Verständigung mit dem darunterliegenden Fenster vorbei. Ein Programmabbruch durch den Anwender ist damit ausgeschlossen. Wenn wir eigene Windows mit der NewWindow-Struktur erstellen, ergeht es uns genauso. Wir können uns nur über den IDCMP mit dem Programm verständigen. Zum Glück ist es bei den Layers nicht so kompliziert. An der Speicherstelle 40 der Layer-Struktur befindet sich ein Zeiger, der auf das dem Layer zugehörige Fenster zeigt (falls vorhanden). Sie können das gut nachvollziehen. Wenn Sie sich gerade im Basic-Fenster befinden, können Sie die beiden folgenden Anweisungen im Direkt-Modus eingeben:

```
PRINT WINDOW(7)
PRINT PEEKL(PEEKL(WINDOW(7)+124)+4Ø)
```

Beide Male erhalten Sie das gleiche Ergebnis, die Struktur des aktiven Fensters. Bei der zweiten Anweisung haben Sie an der 124. Speicherstelle die Struktur des Window-Layers geholt und dessen 40. Speicherstelle ausgelesen. Um das neue Layer dem aktiven Fenster zuzuweisen, gehen wir den umgekehrten Weg. Wir poken einfach die Adresse der Window-Struktur in die 40. Speicherstelle der Layer-Struktur:

```
POKEL lay&+4Ø,win&
```

Jetzt fehlt uns nur noch eines zu unserem Glück, ein RastPort. Die Zeichen-Routinen der Grafik-Library benötigen bekanntermaßen fast immer die Adresse der RastPort-Struktur. Wir finden den Layer-RastPort an der 12. Speicherstelle der Layer-Struktur. Damit haben wir alles zusammen, um ein Scroll-Programm mit Hilfe der Layers zu schreiben. Sicherlich sind Sie auch gespannt, ob sich der Aufwand gelohnt hat.

```
REM ScrollLayer  Pfad: Modus/11Riesengrafik/ScrollLayer
'   oder Ringelspiel Version 2
'P11-3
CLEAR
DEFINT a-z
sh=2ØØ:tiefe=2:sb=64Ø
WINDOW 2,,,Ø
win&=WINDOW(7):scr&=PEEKL(win&+46):rp&=WINDOW(8)
li&=scr&+224:bm&=scr&+184:PALETTE 2,Ø,1,Ø
GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
GOSUB Bitmapanlegen :IF fehl THEN ende
'aktuelle Layer-Dimensionen holen
  Wl&=PEEKL(rp&):lmix=PEEKW(Wl&+16):lmiy=PEEKW(Wl&+18)
  lmax=PEEKW(Wl&+2Ø):lmay=PEEKW(Wl&+22)
'Layer aufrufen
  lay&= CreateUpFrontLayer&(li&,bm&,lmix,lmiy,lmax,lmay,4,mb&)
  IF lay&=Ø THEN fehl=1:GOTO ende
'Layer dem aktuellen Fenster zuweisen
  POKEL lay&+4Ø,win&
  lrp&=PEEKL(lay&+12)

GOSUB zeichnen
txt-= "ENDE = beliebige Taste":tx&=SADD(txt-)
CALL Move&(lrp&,41Ø,2Ø):CALL Text&(lrp&,tx&,22)
FOR i=1 TO bmbr-sb
  CALL ScrollLayer&(li&,lay&,1,Ø)
NEXT i
FOR i=1 TO bmbr-sb
  CALL ScrollLayer&(li&,lay&,-1,Ø)
NEXT i

a=-1
WHILE a
  ta-=INKEY-:IF ta-<>"" THEN a=Ø
WEND
```

```
ende:
IF lay& THEN CALL DeleteLayer&(li&,lay&)
FOR i = 0 TO tiefe-1
  IF bp&(i) THEN CALL FreeRaster&(bp&(i),bmbr,bmRows)
NEXT
IF rk& THEN CALL FreeRemember&(rk&,-1)
WINDOW CLOSE 2
IF fehl THEN
  ON fehl GOSUB f1,f2
  BEEP:PRINT ft-
END IF
LIBRARY CLOSE :END

f1:ft-= "Fehler bei CreateUpFrontLayer":RETURN
f2:ft-= "Speicher nicht ausreichend":RETURN

LibraryOeffnen:
DECLARE FUNCTION AllocRaster&() LIBRARY
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION CreateUpFrontLayer&() LIBRARY
LIBRARY ":bue/layers.library"
LIBRARY ":bue/intuition.library"
LIBRARY ":bue/graphics.library"
RETURN

Speicher:
art&=3+(2^16):rek&=0:rk&=VARPTR(rek&)
mb&=AllocRemember&(rk&,100,art&)'fuer Bitmap-Struktur
IF mb&=0 THEN fehl=2
RETURN

zeichnen:
FOR i=0 TO 50
  x=RND*bmbr:y=RND*sh :x1=RND*sh/2:y1=RND*sh/2
  fa=RND*1+1 'Farbe 3 wegen flimmerns weggelassen
  CALL SetApen&(lrp&,fa)
  CALL DrawEllipse&(lrp&,x,y,x1,y1)
NEXT
RETURN

Bitmapanlegen:
bmbr=1024
bmBytPerRow=bmbr/8      'Bytes per Grafikzeile
```

```
bmRows=sh                  'Grafik-Zeilen
volum&=bmBytPerRow*bmRows
FOR i = Ø TO tiefe-1
  bp&(i)=AllocRaster&(bmbr,bmRows)
  IF bp&(i)=Ø THEN fehl=2:RETURN
  CALL BltClear&(bp&(i),volum&,Ø)
NEXT
POKEW mb&,bmBytPerRow         'Bytes/Reihe
POKEW mb&+2,bmRows            'Reihen
POKE mb&+5,tiefe              'Tiefe
FOR n = Ø TO tiefe-1
  POKEL mb&+8+(n*4),bp&(n)    'n BitPlanes
NEXT
CALL InitBitMap&(mb&,tiefe,bmbr,bmRows)
RETURN
```

Bevor wir das Ergebnis diskutieren, werfen wir erst einen Blick auf den Programmaufbau. Wir behalten den Workbench-Screen bei und öffnen nur ein neues Fenster ohne Titel-Leiste. Anschließend holen wir uns die Start-Adressen der Window-Struktur (*win&*), der Screen-Struktur (*scr&*), des Fenster-RastPorts (*rp&*) und der Standard-Bitmap (*bm&*). In der Subroutine »LibraryOeffnen« deklarieren wir die benötigten Funktionen und öffnen die Layers-, die Intuition- und die Grafik-Bibliothek.

Bei »Speicher« reservieren wir einige Bytes für die neue Bitmap-Struktur. Diese legen wir, wie bereits einige Male praktiziert, in der Subroutine »Bitmapanlegen« an. Die Bitmap soll 1024 Pixel breit und 200 Pixel hoch werden. Wir werden also später im Programm nur um die X-Achse scrollen.

Nun wird's wieder spannend. Für das neue Layer holen wir uns die Layer-Abmessungen des aktuellen WINDOW 2. Bei der Erklärung des Prinzips hatten wir die Adresse der Layer-Struktur aus der 124. Speicherstelle der Window-Struktur geholt. Dieses Mal holen wir sie aus der ersten Speicherstelle der RastPort-Struktur des Windows. Mit den Layer-Dimensionen haben wir alle Parameter zusammen, um *CreateUpFrontLayer* aufzurufen. Wie vorher besprochen, machen wir anschließend aus dem herrenlosen Layer ein Window-Layer des aktuellen Fensters. Mit dem ermittelten Layer-RastPort können wir dann zum »Zeichnen« der Grafik schreiten.

Dazu verteilen wir 51 Ellipsen verschiedener Größe und Farbe zufällig über den Layer-RastPort in die Layer-Bitmap. Als krönenden Abschluß der Grafik setzen wir einen Hinweis an den Anwender so auf den Bildschirm, daß er während des gesamten Scroll-Vorganges sichtbar bleibt. Damit läßt sich recht gut die Breite des Scroll-Bereiches abschätzen.

Damit können wir mit dem Rollen der Grafik beginnen. Mit der Routine *ScrollLayer* scrollen wir die Grafik Bildpunkt für Bildpunkt um die X-Achse hin und wieder zurück. Die zurückgelegte Strecke beträgt *bmbr-sb* = 384 Grafik-Punkte. Anschließend wartet das Programm in einer Schleife auf den Tastendruck des Anwenders, der das Programm beendet. Dazu wird zuerst das Layer gelöscht. Es folgen die Freigaben der Speicherbereiche der Bitmap und für die Bitmap-Struktur. Nachdem das Fenster wieder geschlossen wurde, werden auch die Libraries nicht mehr benötigt.

Wie gefällt Ihnen das Scrollen mit der Layer-Technik? Ich finde, daß der Scroll-Vorgang zwar schön sanft vor sich geht, daß dabei aber leider das Bild flimmert. Es flimmert tatsächlich viel stärker, als das Programm es zeigt. Ich habe nämlich die Farbe 3, die besonders stark flimmert, aus dem Bild herausgelassen. Ein kreisförmiges Scrollen ist mit dieser Technik nicht möglich. Der Vorteil, den es bietet, darf natürlich nicht unterschlagen werden. Immerhin kommen wir ohne MessagePort aus.

Experimentieren Sie noch ein wenig mit dem Programm herum. Probieren Sie das Rollen in der Y-Achse, was durch geringfügige Änderungen am Programm möglich ist. Ich persönlich halte die erste Version mit dem rollenden ViewPort für diejenige von den beiden, die mit weit geringerem Aufwand ein besseres Ergebnis zeigt. Das Programmbeispiel CAVE, welches Sie im Kapitel *Programme* finden, arbeitet daher mit dieser Technik.

Kapitel 12

HAM

Wenn von den Spezial-Modi des Amiga die Rede ist, wird an erster Stelle der HAM-Modus genannt. HAM ist das Zauberwort für die gleichzeitige Darstellung von 4096 verschiedenen Farben. Diese drei Buchstaben lassen die Herzen fast aller Amiga-Freaks höher schlagen. Dem Basic-Programmierer allerdings fällt ein Wermutstropfen in den Becher der Freude. Die Befehle des Standard-Basic ignorieren den HAM-Modus völlig. Damit ist es nun vorbei. Mit unserem Wissen über die System-Software werden wir dem Amiga auch dieses Geheimnis entreißen.

12.1 HAM wird entschlüsselt

HAM ist die Abkürzung für Hold And Modify. Damit ist, zumindest in Kurzform, das Prinzip der Darstellung bereits erklärt. Fangen wir aber lieber von vorne an. Normalerweise haben wir zur farblichen Gestaltung maximal 5 BitPlanes bei einer Farbtiefe 5 zur Verfügung. Damit lassen sich bekanntermaßen 2^5 = 32 verschiedene Farben auf den Bildschirm bringen. Der HAM-Modus arbeitet mit 6 BitPlanes. Das entspräche also einer Farbtiefe 6. Die Rechnung 2^6 ergibt aber 64. Damit könnten dann auch nur 64 Farben gezeigt werden. Wie soll es dann zur gleichzeitigen Darstellung von 4096 verschiedenen Farben kommen? Eigentlich braucht man dazu die doppelte Anzahl, nämlich 12 Bit-Ebenen! Haben Sie einmal nachgerechnet, welchen Speicherplatz ein solcher Super-Screen beanspruchen würde? Richtig! Etwa 122 Kbyte würde alleine der Bildschirm verschlingen. Bei diesem enormen Speicherbedarf ist die Frage, ob die Hardware des Amiga diese 12 Bit-Ebenen überhaupt verwalten könnte, nebensächlich.

Um mit den genannten 6 BitPlanes auszukommen, sind die Schöpfer des Amiga einen anderen Weg gegangen. Wenn ein Bildpunkt gesetzt werden soll, wird der links von dem zu setzenden Pixel liegende Punkt zur Farbwahl herangezogen. Die obersten beiden Bit-Ebenen, also die Planes 5 und 6 des zu setzenden Bildpunktes, bestimmen, welcher Farbbestandteil, rot, grün oder blau, des links liegenden Pixels geändert bzw. modifiziert werden soll, um die neue Farbe zu erhalten. Die unteren 4 Bit bestimmen dabei die Intensität des zu modifizierenden Farbbestandteils.

Uff, das war ein harter Brocken. Auf Anhieb ist das Prinzip sicherlich nur sehr schwer zu verstehen. Diese Farbgebung widerspricht allem, was Sie sich bisher darüber erarbeitet haben. Das beste ist deshalb, Sie trennen sich gedanklich von diesem Wissen, wenn Sie im HAM-Modus programmieren. Ich will Ihnen beileibe keine Angst einjagen. Wir werden die Sache so detailliert angehen, daß Sie das Prinzip im Schlaf beherrschen können. Es folgen außerdem fünf Programmbeispiele, die Ihnen die Programmierung des HAM-Modus in der Praxis demonstrieren werden. Schauen wir uns nun die Bit-Ebenen 5 und 6 an, die für die Auswahl des zu ändernden Farbbestandteiles (r,g,b) zuständig sind.

Die Bit-Kombination 00
besagt, daß der links liegende Bildpunkt *nicht* zur Farbwahl herangezogen wird. Die Bits der Ebenen 1 bis 4 setzen die Farbe wie gewohnt zusammen. Diese Kombination können Sie also zum Setzen eines Punktes, dessen rechts folgenden Punkt Sie anschließend modifizieren wollen, verwenden. Diese 16 möglichen Farben haben die Farbnummern 0 bis 15. Sie setzen die Farbe, indem Sie den Zeichenstift mit der Grafik-Routine *SetAPen* aufrufen.

Die Bit-Kombination 01
legt fest, daß von dem links liegenden Grafikpunkt die Farbbestandteile Rot und Grün gehalten werden. Der blaue Farbanteil erhält die Intensität der vier Bits 0–3. Wählen Sie zum Beispiel den maximalen Blauanteil 15 ($f), so müssen Sie die Farbnummer 15 plus das gesetzte 4. Bit = 16 (2^4) angeben. Die Farbnummer wäre dabei 31. Es sind die Farben 16 bis 31 möglich. Um den Farb-Modus sofort erkennen zu können, ist es besser und übersichtlicher, im Programm die Zahl getrennt anzugeben (15+16). Schauen wir uns das Ganze einmal an:

Farbe Pixel links	**Befehl für neues Pixel**	**Farbe neues Pixel**
&h0af0 (hellgrün) r($a)u.g($f) halten	CALL SetAPen&(rp&,15+16) mit blau($f) modifizieren	&h0aff (hellblau) ergibt rgb($aff)
&h0ff0 (gelb) r($f)u.g($f) halten	CALL SetAPen&(rp&,14+16) mit blau($e) modifizieren	&h0ffe (weiß) ergibt rgb($ffe)

Die Bit-Kombination 10
sagt aus, daß die Farbbestandteile Grün und Blau des Pixels, welches links von dem zu setzenden Grafikpunktes liegt, gehalten werden. Der Wert der Bit-Kombination ist 32 (2^5). Der rote Farbanteil setzt sich aus den Bits der Ebenen 1–4 zusammen. Die Farbnummern 32 bis 47, oder besser ausgedrückt 32+0 bis 32+15, sind möglich. Die Tabelle zeigt, wie das praktisch aussehen kann:

Farbe Pixel links	Befehl für neues Pixel	Farbe neues Pixel
&h009b g(9)u.b($b) halten	CALL SetAPen&(rp&,7+32) mit rot(7) modifizieren	&h079b ergibt rgb($79b)
&h0019 g(1)u.b(9) halten	CALL SetAPen&(rp&,12+32) mit rot($c) modifizieren	&h0c19 ergibt rgb($c19)

Die Bit-Kombination 11
bestimmt, daß die roten und blauen Farbbestandteile des linken Bildpunktes gehalten werden. Die Bit-Kombination ist an ihrem Wert 48 (2^4+2^5) zu erkennen. Der grüne Farbanteil erhält seine Intensität aus den Bits der Ebenen 1 bis 4. Die Farbnummern 48 bis 63, besser erkennbar als 48+0 bis 48+15, können bei der vierten und letzten Bit-Kombination ausgewählt werden. Auch wenn Sie inzwischen das Prinzip durchschaut haben, soll die folgende Tabelle bei auftretenden Unsicherheiten helfen:

Farbe Pixel links	Befehl für neues Pixel	Farbe neues Pixel
&h0b02 r($b)u.b(2) halten	CALL SetAPen&(rp&,10+48) mit grün($a) modifizieren	&h0ba2 ergibt rgb($ba2)
&h040a r(4)u.b($a) halten	CALL SetAPen&(rp&,3+48) mit grün(3) modifizieren	&h043a ergibt rgb($43a)

12.2 HAM aktivieren

Nachdem Sie nun ganz genau wissen, wie die Bildpunkte durch *halten* und *modifizieren* gesetzt werden, brennen Sie sicherlich schon darauf, den HAM-Modus in Aktion zu erleben. Überlegen wir uns, was wir dazu alles benötigen. HAM ist ein View-Modus wie Hires, Lores oder jeder andere Modus auch. Für den HoldAndModify-Modus muß in der 32. Speicherstelle der ViewPort-Struktur das Bit 11 gesetzt werden. Das 11. Bit wird durch den Wert 2^11=2048 repräsentiert.

Wir brauchen also nur eine NewScreen-Struktur zu erstellen, in die wir den View-Modus HoldAndModify und eine Farbtiefe von 6 eintragen werden. Aus der NewScreen-Struktur zaubert die Intuition einen neuen Bildschirm mit allem, was dazugehört. Mit dem neu erstellten Screen könnten wir bereits arbeiten. Damit aber das Programm mit dem Anwender über den IDCMP korrespondieren kann, benötigen wir zusätzlich ein neues Fenster mit einem oder mehreren benötigten IDCMP-Flags. Das ist schon alles. Daß wir zum Zeichnen die Routinen der Grafik-Library verwenden werden, ist für Sie inzwischen so selbstverständlich, daß der Umstand eigentlich keiner besonderen Erwähnung mehr bedarf.

12.3 4096 Farben auf einen Streich

Wie schon die Überschrift besagt, halten wir uns bei unserem ersten Programm nicht mit Kleinigkeiten auf. Wenn im HAM-Modus die 4096 möglichen Farben gleichzeitig dargestellt werden können, so wollen wir das auch sehen. Für jede der 4096 Farben zeichnet das Programm *colorful* ein kleines Rechteck von 5 Pixel Breite und 4 Bildpunkten Höhe. Mit diesen 20 Pixel, die für jede Farbe zur Verfügung stehen, ist der Bildschirm bis auf den letzten Bildpunkt ausgenutzt.

```
REM colorful  Pfad: Modus/12HAM/colorful
'P12-1
CLEAR
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
sb=320:tiefe=6:vm=2048:tp=15:stit&=0:sbm&=0
wb=sb:wh=sh:idf&=8:fl&=4096
t2-="COLORFUL    Mausklick=ende"+CHR-(0):wtit&=SADD(t2-)
GOSUB LibraryOeffnen
GOSUB Speicher     :IF fehl THEN ende
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,sbm&) :IF fehl THEN ende
CALL Fenster (wb,wh,idf&,fl&,wtit&,tp) :IF fehl THEN ende

'Bit-Ebenen 5 und 6 > 00 > 0  > Standard aus Bitmap 1-4
'Bit-Ebenen 5 und 6 > 01 > 16 > Blauwert aus Bitmap 1-4
'Bit-Ebenen 5 und 6 > 10 > 32 > Rotwert  aus Bitmap 1-4
'Bit-Ebenen 5 und 6 > 11 > 48 > Gruenwert aus Bitmap 1-4

CALL SetRGB4&(vp&,0,0,0,0)  'Hintergrundfarbe schwarz!!!
tm&=TIMER:WHILE TIMER<tm&+5:WEND
x=0:y=0:bw=4:IF sh>255 THEN bh=3 ELSE bh=2
FOR r = 0 TO 15
  FOR g = 0 TO 15
    FOR b = 0 TO 15
  CALL SetAPen&(rp&,r+32) 'haelt Pixel lks.(0), modif. rot>r00
  CALL Move&(rp&,x,y)
  CALL Draw&(rp&,x,y+bh)
  CALL SetAPen&(rp&,g+48) 'haelt Pixel lks(r00), modif.gr>rg0
  CALL Move&(rp&,x+1,y)
  CALL Draw&(rp&,x+1,y+bh)
  CALL SetAPen&(rp&,b+16) 'haelt Pixel lks(rg0), modif.bl>rgb
  CALL RectFill&(rp&,x+2,y,x+bw,y+bh)
  x=x+bw+1:IF x>315 THEN x=0:y=y+bh+1
NEXT b,g,r
```

```
a=-1
WHILE a
  me&=PEEKL(win&+86)
  abfrage: mess&=GetMsg&(me&):IF mess&=Ø THEN abfrage
  inm&=PEEKL(mess&+2Ø)     'Class of IntuitionMessage
  CALL ReplyMsg&(mess&)
  IF inm&=8 THEN a=Ø       'MOUSEBUTTONS = 8
WEND

ende:
IF win& THEN CALL CloseWindow&(win&)
IF scr& THEN CALL CloseScreen&(scr&)
IF rk& THEN CALL FreeRemember&(rk&,-1)
IF fehl THEN
  ON fehl GOSUB F1,F2,F3
  BEEP:PRINT ft-
END IF
LIBRARY CLOSE :END

F1:ft-= "Fehler bei OpenWindow":RETURN
F2:ft-= "Speicher nicht ausreichend":RETURN
F3:ft-= "FEHLER bei OpenScreen":RETURN

LibraryOeffnen:
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION OpenScreen&() LIBRARY
DECLARE FUNCTION OpenWindow&() LIBRARY
DECLARE FUNCTION GetMsg&() LIBRARY
LIBRARY ":bue/intuition.library"
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/exec.library"
RETURN

Speicher:
art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
ms&=AllocRemember&(rk&,4Ø,art&)
IF ms&=Ø THEN fehl=2
mw&=AllocRemember&(rk&,48,art&)
IF mw&=Ø THEN fehl=2
RETURN
```

```
SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
SHARED ms&,scr&,rp&,mb&,vp&,fehl%
StrukturNeuerScreen:
POKEW ms&,Ø       :POKEW ms&+2,Ø      'linke u.obere Kante
POKEW ms&+4,sb% :POKEW ms&+6,sh%    'Breite, Hoehe
POKEW ms&+8,tie%                      'Tiefe
POKE ms&+1Ø,Ø    :POKE ms&+11,1      'DetailPen, BlockPen
POKEW ms&+12,vm%                      'ViewModes  HAM
POKEW ms&+14,tp%                      'Type  CUSTOMSCREEN
POKEL ms&+16,Ø                        'TextAttr
POKEL ms&+2Ø,stit&                    'Screen-Titel
POKEL ms&+24,Ø   :POKEL ms&+28,sbm& 'Gadget,Bitmap
scr&= OpenScreen&(ms&)
IF scr&=Ø THEN fehl%=3
rp&=scr&+84 :mb&=PEEKL(rp&+4) :vp&=scr&+44
END SUB

SUB Fenster (wb%,wh%,idf&,fl&,wtit&,tp%) STATIC
SHARED mw&,scr&,win&,fehl%
StrukturNeuesFenster:
POKEW mw&,Ø       :POKEW mw&+2,Ø     'linke u.obere Kante
POKEW mw&+4,wb% :POKEW mw&+6,wh%   'Breite, Hoehe
POKE mw&+8,1     :POKE mw&+9,Ø      'DetailPen, BlockPen
POKEL mw&+1Ø,idf&                    'IDCMPFlag MOUSEBUTTONS
POKEL mw&+14,fl&                     'Flags ACTIVATE
POKEL mw&+26,wtit&                   'Window-Titel
POKEL mw&+3Ø,scr&                    'Screen
POKEW mw&+38,1ØØ:POKEW mw&+4Ø,3Ø   'min. Breite und Hoehe
POKEW mw&+42,wb%:POKEW mw&+44,wh%  'max. Breite und Hoehe
POKEW mw&+46,tp%                     'Type  CUSTOMSCREEN
win&= OpenWindow&(mw&)
IF win&=Ø THEN fehl%=1
END SUB
```

Zuerst legen wir die einzelnen Variablen für den neuen Screen und das neue Window fest. In der Variablen *sh* wird die Höhe des Workbench-Screens festgehalten. Das Programm berücksichtigt *sh* entsprechend. Es läuft also mit unterschiedlichen Screen-Höhen. Anschließend werden die Libraries geöffnet und die Speicherbereiche für die NewScreen- und NewWindow-Strukturen belegt. Damit können wir nun das Unterprogramm »Schirm« aufrufen. Wichtig bei der NewScreen-Struktur sind die drei Felder für den View-Modus (HAM=2048), die Screen-Type (CUSTOMSCREEN=15) und für die Tiefe (6). Auch das Unterprogramm »Fenster« wird nun aufgerufen. Besonders

erwähnenswert in der NewWindow-Struktur sind die Felder für das IDCMP-Flag (MOUSEBUTTONS=8) und die Fenster-Type (CUSTOMSCREEN=15). Die beiden genannten Unterprogramme sind so ausgelegt, daß sie auch für die anderen Programme des Kapitels eingesetzt werden können.

Zur Information für den Anwender folgt im Programm eine kurze Zusammenfassung des Prinzips der Farbgestaltung im HAM-Modus. Nun wird es langsam interessant. Mit *SetRGB4* wird die Hintergrundfarbe auf Schwarz gesetzt. Die drei Ausrufezeichen beim Kommentar sollen ausdrücken, wie wichtig diese einfache Tatsache für das weitere Verständnis des Programmes ist. Anschließend wartet das Programm 5 Sekunden in einer WHILE/WEND-Schleife, bis der Anwender den Text des Fenster-Titels gelesen hat. Das ist deshalb wichtig, weil die Titel-Zeile gleich überschrieben wird und der Anwender wissen soll, wie er das Programm beenden kann.

In den drei ineinandergeschachtelten FOR/NEXT-Schleifen werden die 4096 verschiedenfarbigen Rechtecke gezeichnet. Die äußerste Schleife setzt den Rotwert, die mittlere den Grünwert und die innere Schleife den blauen Farbanteil. Die innere Schleife betrachten wir uns nun ganz genau. Mit

```
CALL SetAPen&(rp&,r+32)
```

rufen wir die Bit-Kombination *10* der Ebenen 5 und 6 auf. Es wird das linke Pixel gehalten und mit dem Rotwert *r* der inneren Schleife modifiziert. Da wir als Hintergrundfarbe Schwarz gewählt haben, werden die Farbanteile Grün und Blau jeweils mit dem Wert Null gehalten. Wir erhalten dadurch einen sauberen Farbaufbau für die Ausgabe aller Farben. Deshalb auch die drei Ausrufezeichen bei *SetRGB4*. Mit der modifizierten Farbe ziehen wir nun eine kurze senkrechte Linie mit *Move* und *Draw*. Die zweite Modifikation,

```
CALL SetAPen&(rp&,g+48)
```

mit der Bit-Kombination *11* der BitPlanes 5 und 6, hält von dem Bildpunkt links die Farbanteile Rot und Blau und modifiziert ihn mit dem Grünwert *g* der mittleren Schleife. Da wir die zuletzt gezogene Linie aber nur mit dem Farbanteil von Rot gezeichnet haben, erhalten wir wieder eine saubere Mischung der Farbbestandteile Rot und Grün. Mit dieser Farbmischung ziehen wir nun die zweite senkrechte Linie mit den Routinen der Grafik-Bücherei. Kommen wir nun zur dritten Modifikation mit der Bit-Kombination *01* der BitPlanes 5 und 6. Die Routine

```
CALL SetAPen&(rp&,b+16)
```

hält vom linken Bildpunkt die Farbbestandteile Rot und Grün und modifiziert die Farbmischung mit dem blauen Wert der äußeren Schleife. Damit haben wir endlich die gewünschte Farbzusammensetzung erreicht und können mit *RectFill* das Rechteck zeichnen. Am Ende der inneren Schleife wird die Variable *x* um 5 Pixel erhöht, damit wir für den nächsten Durchlauf, wieder vom schwarzen Wert der Hintergrundfarbe beginnend, eine neue Farbe mischen können.

Der Rest ist schnell erklärt. Das Programm fragt fortwährend in der WHILE/WEND-Schleife den Intuition-MessagePort ab, ob die linke Maustaste gedrückt wurde. Sobald dieses Ereignis eintritt, wird die Schleife verlassen und das Programm mit den Schließ-Routinen und Speicherfreigaben ordentlich beendet.

12.4 HAM mit 5 Bit-Ebenen

Sie haben bisher immer gelesen, daß Sie für den HAM-Modus 6 BitPlanes benötigen. Nun, es funktioniert auch mit nur 5 Bit-Ebenen. Allerdings sind damit die Möglichkeiten dieses Super-Modus stark eingeschränkt. Da die 6. Bit-Ebene entfällt, haben Sie dann nur die beiden Bit-Kombinationen 00 und 01 zur Verfügung. Mit der Bit-Kombination 00 wählen Sie die Standard-Farben 0 bis 15. Die Bit-Kombination 01 der Ebene 5 läßt eine Modifizierung des Blauwertes mit dem Wert der unteren 4 BitPlanes zu, während von dem links liegenden Grafikpunkt die Farbbestandteile Rot und Grün gehalten werden. Es sind die modifizierten Farben 16+0 bis 16+15 möglich. Die Anzahl der gleichzeitig auf dem Bildschirm darstellbaren Farben liegt deutlich niedriger und ist auf maximal 256 begrenzt.

Die programmtechnische Ausführung unterscheidet sich kaum zum HAM-Modus mit 6 Bit-Ebenen. Auch hier müssen Sie nur eine NewScreen-Struktur erstellen, in die der View-Modus *HoldAndModify* mit einem Wert von 2048 eingetragen wird. Der einzige Unterschied besteht darin, daß Sie beim eingeschränkten HAM-Modus eine Farbtiefe von 5 eintragen.

Das Programm *MiniHAM* demonstriert, wie die Farben des HAM-Modus mit 5 Bit-Ebenen aussehen können. Kleine Rechtecke und Kreise, durch Zufallszahlen im Fenster verstreut, werden in den Standard-Farben gezeichnet und anschließend jeweils 9 Mal mit dem blauen Farbanteil modifiziert.

```
REM MiniHAM  Pfad: Modus/12HAM/MiniHAM
REM P12-2
CLEAR
DEFINT a-z :DIM f(15)
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
sb=320:tiefe=5:tp=15:stit&=0:sbm&=0
vm=2048               'View-Modus HoldAndModify
wb=sb:wh=sh:idf&=8:fl&=4096
t2-="HAM mit 5 BitMaps   Mausklick=ende"+CHR-(0):wtit&=SADD(t2-)
GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,sbm&) :IF fehl THEN ende
CALL Fenster (wb,wh,idf&,fl&,wtit&,tp) :IF fehl THEN ende
GOSUB farben
GOSUB zeichnen
```

```
a=-1
WHILE a
  me&=PEEKL(win&+86)
  abfrage: mess&=GetMsg&(me&):IF mess&=Ø THEN abfrage
  inm&=PEEKL(mess&+2Ø)      'Class of IntuitionMessage
  CALL ReplyMsg&(mess&)
  IF inm&=8 THEN a=Ø        'MOUSEBUTTONS = 8
WEND

ende:
  IF win& THEN CALL CloseWindow&(win&)
  IF scr& THEN CALL CloseScreen&(scr&)
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN
    ON fehl GOSUB f1,F2,F3
    BEEP:PRINT ft-
  END IF
  LIBRARY CLOSE
END

f1:ft-= "Fehler bei OpenWindow":RETURN
F2:ft-= "Speicher nicht ausreichend":RETURN
F3:ft-= "FEHLER bei OpenScreen":RETURN

farben:
  FOR i=Ø TO 15:READ f(i):NEXT
  CALL LoadRGB4&(vp&,VARPTR(f(Ø)),16)
RETURN

zeichnen:
  r=1Ø:f1=7:d=f1-1
  FOR i = Ø TO 3Ø
    RANDOMIZE TIMER
    y=(wh-3Ø)*RND+15:x=(wb-3Ø)*RND+2:fa=RND*16-1
    CALL SetAPen&(rp&,fa)
    CALL RectFill&(rp&,x,y,x+1Ø,y+1Ø)
    FOR f=f1 TO 15
      CALL SetAPen&(rp&,f+16)
      CALL RectFill&(rp&,x+f-d,y,x+f-d+1Ø,y+1Ø)
    NEXT f
    RANDOMIZE TIMER
    y=(wh-3Ø)*RND+15:x=(wb-4Ø)*RND+2Ø:fa=RND*16-1
    CALL SetAPen&(rp&,fa)
```

```
    CALL DrawEllipse&(rp&,x,y,r,r)
    FOR f=f1 TO 15
      CALL SetAPen&(rp&,f+16)
      CALL DrawEllipse&(rp&,x+f-d,y,r,r)
    NEXT f
  NEXT
RETURN

LibraryOeffnen:
### bitte hier einfügen ###
RETURN

Speicher:
### bitte hier einfügen ###
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
### bitte hier einfügen ###
END SUB

SUB Fenster (wb%,wh%,idf&,fl&,wtit&,tp%) STATIC
### bitte hier einfügen ###
END SUB

bunt:
DATA &H0fff,&H09c0,&H0af0,&h09b0,&H0d00,&h0ff0,&h0f00,&h0cc0
DATA &h0000,&h0900,&h0190,&h0740,&h00b0,&h04a0,&h0b20,&h0880
```

Vier Programme aus diesem Kapitel sind ein Beispiel dafür, wie man sich durch einheitliche Subroutinen und Unterprogramme die Programmierung leichter machen kann. Sie setzen alle die beiden Subroutinen »Speicher« und »LibraryOeffnen« und die beiden Unterprogramme »Schirm« und »Fenster« ein. Sie kennen sie bereits aus dem letzten Unterkapitel und sie sind daher in diesem und in den folgenden Programmen nicht mehr ausgedruckt.

Bis auf die Dimensionierung der Feld-Variablen *f()* für die 16 Standard-Farben ist Ihnen der Programmanfang vom Vorprogramm geläufig. Das Unterprogramm »Schirm« wird mit einer *tiefe* von 5 aufgerufen, da wir ja den HAM-Modus mit 5 Bit-Ebenen zeigen wollen. In der Subroutine »farben« werden die 16 Standard-Farben eingelesen und mit *SetRGB4* die neue Farbtafel gesetzt. Wenn Sie sich die einzelnen Daten der Farben ansehen, werden Sie feststellen, daß alle keinen blauen Farbanteil bzw. den Wert Null haben. Damit kann die Modifizierung mit dem blauen Farbbestandteil besser gezeigt werden.

Die Darstellung der modifizierten Grafiken erfolgt in der Subroutine »zeichnen«. Die äußere Schleife wird 31 Mal durchlaufen und dabei nacheinander jeweils ein Rechteck und eine kleine Röhre gezeichnet. Mit *SetAPen* wird die Standardfarbe gesetzt und mit *RectFill* ein Rechteck gezeichnet. Nun werden neben das Rechteck, im Abstand von einem Bildpunkt, 9 weitere Rechtecke mit den modifizierten Blauanteilen gezeichnet. Auch die Kreise werden nach der gleichen Methode auf den Bildschirm gebracht. Nachdem der erste Kreis mit der zufällig ermittelten Standardfarbe gezeichnet wurde, werden mit *SetAPen* und der Bit-Kombination 01=16 nacheinander die Bit-Kombinationen gesetzt. Dabei wird immer ein Kreis an den anderen gesetzt (linkes Pixel), damit der blaue Farbanteil modifiziert werden kann.

Nun wartet das Programm wieder in einer WHILE/WEND-Schleife auf einen Mausklick des Anwenders. Mit diesem Klick beenden die Aufräumungsarbeiten das Programm.

12.4.1 HAM in einem Standard-Screen

Wenn wir schon den HAM-Modus mit nur 5 Bit-Ebenen programmieren, müßte es doch möglich sein, dafür einen normalen Screen des Standard-Basic einzusetzen. Natürlich hat für einen ordentlichen Programmierer eine solche Manipulation nur einen Sinn, wenn der Schirm so aufgebaut wird, daß dadurch keine Einschränkung bei den Befehlen eintritt. Wenn wir vergleichen, was einen Standard-Screen der Tiefe 5 von einem Screen im HAM-Modus unterscheidet, bleibt nur das Flag für den View-Modus als Unterschied.

Unser Problem liegt also darin, den HAM-Modus in die Screen-Struktur zu praktizieren und den Rest der Video-Ausgabe entsprechend anzupassen. Wie Sie sicherlich noch aus dem Kapitel über die Video-Ausgabe wissen, ist der ViewPort die Grundlage des Intuition-Screens. Wir schreiben daher den neuen Modus, in unserem Fall HAM, in Wortlänge in die 32. Speicherstelle des ViewPorts. Das alleine genügt natürlich noch nicht. Es müssen noch die einzelnen Grafik-Ausgabeelemente darauf abgestimmt werden. Dazu stellt uns die Intuition die Routine *RemakeDisplay* zur Verfügung. Das folgende Unterprogramm führt die Änderung des View-Modus durch.

```
SUB vpMode (mode%) STATIC
  SHARED vp&
    IF PEEKW(vp&+32) AND mode% THEN     'Modus ausschalten
       POKEW vp&+32,PEEKW(vp&+32) AND NOT mode%
    ELSE                                 'Modus einschalten
       POKEW vp&+32,PEEKW(vp&+32) OR mode%
    END IF
  CALL RemakeDisplay&
END SUB
```

Dem Unterprogramm wird mit dem Parameter *mode%* des View-Modus aufgerufen. Damit haben Sie ein Werkzeug (tool) zur Verfügung, mit dem Sie jeden beliebigen View-Modus einschalten können! Die Adresse der ViewPort-Struktur muß im Programm vorhanden sein, damit (über die SHARED-Anweisung) das Unterprogramm damit arbeiten kann. Zuerst wird geprüft, ob der Modus bereits eingeschaltet ist. In diesem Fall nimmt das Programm an, daß der Anwender den Modus löschen will und schaltet ihn aus. Im anderen Fall wird der neue Modus in die Speicherstelle 32 der ViewPort-Struktur geschrieben. Zum Schluß bringt die Intuition-Routine die Grafik-Ausgabe in Ordnung. Natürlich muß im Hauptprogramm die Intuition-Bibliothek vorher geöffnet werden.

Die einfache Prozedur der Modus-Änderung funktioniert aber genauso sicher, wenn in der Screen-Struktur der View-Modus geändert wird. Sie sehen, es stehen Ihnen sogar zwei Wege zur Auswahl. In dem folgenden Programm wird der View-Modus über die Screen-Struktur geändert. Das ist wirklich schon alles, was zu tun ist, wenn Sie den auf 5 BitPlanes eingeschränkten HAM-Modus für Ihre Programme einsetzen wollen.

Das folgende Programm kommt daher mit einer einzigen Library-Routine aus. Alles andere sind Befehle des Standard-Basic.

```
REM MiniHAM-Standard  Pfad: Modus/12HAM/MiniHAMStd
REM P12-3
CLEAR
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
LIBRARY ":bue/intuition.library"
SCREEN 1,32Ø,sh,5,1
WINDOW 2,"bitte Taste druecken",,1,1
scr&=PEEKL(WINDOW(7)+46)
CALL vpMode (2Ø48)
GOSUB zeichnen

a=-1
WHILE a
  ta$=INKEY$:IF ta$<>"" THEN a=Ø
WEND

ende:
  CALL vpMode (2Ø48)
  WINDOW CLOSE 2
  SCREEN CLOSE 1
```

```
zeichnen:
  PALETTE Ø,.6,.6,.6  'Hintergrundfarbe grau
  y1=WINDOW(3)/2
  FOR xb=-278 TO Ø STEP 19
    mo=16:x=xb
    fa=fa+1
    CIRCLE (x,y1),3ØØ,fa,5.9,.39
    FOR i=9 TO 15
      x=x+1
      CIRCLE (x,y1),3ØØ,i+mo,5.9,.39
    NEXT
    FOR i=15 TO 9 STEP-1
      x=x+1
      CIRCLE (x,y1),3ØØ,i+mo,5.9,.39
    NEXT
  NEXT
RETURN

SUB vpMode (mode%) STATIC
  SHARED scr&
  rp&=WINDOW(8)
    IF PEEKW(scr&+76) AND mode% THEN     'Modus ausschalten
       POKEW scr&+76,PEEKW(scr&+76) AND NOT mode%
    ELSE                                 'Modus einschalten
       POKEW scr&+76,PEEKW(scr&+76) OR mode%
    END IF
  CALL RemakeDisplay&
END SUB
```

Nachdem die Intuition-Library, die nur zur Änderung des View-Modus eingesetzt wird, geöffnet ist, werden ein Screen mit 5 Bit-Ebenen und ein Fenster geöffnet. Aus der Window-Struktur wird die Adresse der Screen-Struktur ermittelt. Damit kann bereits das Unterprogramm »vpMode« aufgerufen werden. Dadurch steht der HAM- Modus für die folgenden Grafik-Anweisungen zur Verfügung.

In der Subroutine »zeichnen« werden 15 Kreisbögen in den Standard-Farben, also mit dem Bit-Wert Null, gezeichnet. Diese Kreisbögen werden mit dem Bit-Wert 16 im blauen Anteil modifiziert. Dazu müssen die Kreise natürlich aneinanderliegen (linkes Pixel). Ist die Grafik fertiggestellt, kann der Anwender das Programm mit einem Tastendruck beenden. Jetzt wird mit einem zweiten Aufruf des Unterprogrammes »vpMode« der

12.5 BLUE BOX und 3-D-Zylinder

Inzwischen wissen Sie ganz genau, wie im HAM-Modus gleichzeitig 4096 verschiedene Farben auf den Bildschirm gebracht werden. Das ist zwar eine prima Sache, aber für ein normales Basic-Programm tun es sicher ein paar Farben weniger auch. Interessant wären aber auch Grafiken, die Ton in Ton gezeichnet werden. So könnte eine Grafik aus lauter roten oder grünen Farbtönen optisch recht eindrucksvoll anzusehen sein. Wie können solche zusammenhängende Farben am besten programmiert werden? Um zum Beispiel lauter rote Farbtöne zusammenzustellen, genügt es nicht, bei der Farbe nur den roten Farbanteil zu versorgen. Das ergibt maximal 15 Rottöne, von denen die Farben mit einem geringen Rotanteil auch noch schwarz ausgegeben werden. Im Zweifelsfall hilft dann nur probieren. Vielleicht hilft es aber auch, wenn Sie sich das Programm *colorful* und sein Ergebnis einmal genauer ansehen. Sie sehen dabei eine Reihe zusammenhängender Farbtöne, die mit unterschiedlicher Intensität immer wieder in den einzelnen Schleifen ausgegeben werden. Das folgende Programm ist ein Beispiel für diese Art der Farbwahl. Es zeichnet ein Quadrat, das aus 180 verschiedenen Blautönen gezeichnet wird. Die Farbauswahl erfolgt dabei, wie gerade beschrieben, in reduzierten Schleifen des Programmes *colorful.*

```
REM BLUE BOX  Pfad: Modus/12HAM/blau
'P12-4
'18Ø Blautoene
CLEAR
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
sb=32Ø:tiefe=6:vm=2Ø48:tp=15:stit&=Ø:sbm&=Ø
wb=sb:wh=sh:idf&=8:fl&=4Ø96
t2-="BLUE BOX    Mausklick=ende"+CHR-(Ø):wtit&=SADD(t2-)
GOSUB LibraryOeffnen
GOSUB Speicher     :IF fehl THEN ende
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,sbm&) :IF fehl THEN ende
CALL Fenster (wb,wh,idf&,fl&,wtit&,tp) :IF fehl THEN ende

CALL SetRGB4&(vp&,Ø,15,Ø,Ø)  'Hintergrundfarbe rot
box=18Ø:x=(sb-box)/2:y=(sh-box)/2
FOR b = 7 TO 15
  FOR g = 3 TO Ø STEP -1
    FOR r = 4 TO Ø  STEP -1
      CALL SetAPen&(rp&,r+32)
      CALL Move&(rp&,x,y)
      CALL Draw&(rp&,x,y)
      CALL SetAPen&(rp&,g+48)
```

```
      CALL Move&(rp&,x+1,y)
      CALL Draw&(rp&,x+1,y)
      CALL SetAPen&(rp&,b+16)
      CALL Move&(rp&,x+2,y)
      CALL Draw&(rp&,x+box,y)
      y=y+1
NEXT r,g,b

a=-1
WHILE a
  me&=PEEKL(win&+86)
  abfrage: mess&=GetMsg&(me&):IF mess&=Ø THEN abfrage
  inm&=PEEKL(mess&+2Ø)     'Class of IntuitionMessage
  CALL ReplyMsg&(mess&)
  IF inm&=8 THEN a=Ø       'MOUSEBUTTONS = 8
WEND

ende:
IF win& THEN CALL CloseWindow&(win&)
IF scr& THEN CALL CloseScreen&(scr&)
IF rk& THEN CALL FreeRemember&(rk&,-1)
IF fehl THEN
  ON fehl GOSUB F1,F2,F3
  BEEP:PRINT ft-
END IF
LIBRARY CLOSE :END

F1:ft-= "Fehler bei OpenWindow":RETURN
F2:ft-= "Speicher nicht ausreichend":RETURN
F3:ft-= "FEHLER bei OpenScreen":RETURN

LibraryOeffnen:
### bitte hier einfügen ###
RETURN

Speicher:
### bitte hier einfügen ###
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
### bitte hier einfügen ###
END SUB

SUB Fenster (wb%,wh%,idf&,fl&,wtit&,tp%) STATIC
### bitte hier einfügen ###
END SUB
```

Zum Programm selbst ist sicher nicht viel zu sagen. Die FOR/NEXT- Schleifen sorgen für das Zeichnen der 180 blauen Linien. Die äußerste Schleife beschränkt sich auf die Farbintensität 7 bis 15 des blauen Anteils. Die beiden inneren Schleifen mischen grüne und rote Anteile geringer Intensität dazu. Damit die Farbmischungen in der richtigen Reihenfolge gezeichnet werden, arbeiten diese Schleifen mit fallenden Werten. Wie bereits besprochen, sind die 4 Subroutinen und Unterprogramme nicht abgedruckt.

Kommen wir nun zu einer interessanten Nebenerscheinung bei der Modifizierung von Farben. Das langsame Steigen oder Fallen der Farbtöne oder der Übergang in benachbarte Farben müßte doch nutzbringend einzusetzen sein. Selbst bei oberflächlicher Betrachtung erkennt man schnell, daß sich damit kinderleicht Lichtspiegelungen und Schattenwirkungen zeichnen lassen. Das ist ja genau das, was wir bei der dreidimensionalen Darstellung von Körpern verzweifelt suchen. Das folgende Programm nutzt diesen Effekt aus. Ohne besonderen Aufwand werden zylindrische Körper räumlich dargestellt.

```
REM Zylinder  Pfad: Modus/12HAM/Zylinder
'P12-5
'raeumliche Darstellung zylindrischer Koerper
CLEAR
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
sb=32Ø:tiefe=6:vm=2Ø48:tp=15:stit&=Ø:sbm&=Ø
wb=sb:wh=sh:idf&=8:fl&=4Ø96
t2-="3D-Zylinder    Mausklick=ende"+CHR-(Ø):wtit&=SADD(t2-)
GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,sbm&) :IF fehl THEN ende
CALL Fenster (wb,wh,idf&,fl&,wtit&,tp) :IF fehl THEN ende
CALL SetRGB4&(vp&,Ø,12,12,12)  'Hintergrundfarbe grau

y1=1Ø:y2=sh-1
FOR xb=1Ø TO 3ØØ STEP 15
  mo=RND*2+1:mo=mo*16 :x=xb
  CALL SetRGB4&(vp&,1,Ø,Ø,Ø)
  CALL SetAPen&(rp&,1)
  CALL Move&(rp&,x,y1)
  CALL Draw&(rp&,x,y2)
  FOR i=11 TO 15
    x=x+1
    CALL SetAPen&(rp&,i+mo)
    CALL Move&(rp&,x,y1)
```

```
    CALL Draw&(rp&,x,y2)
  NEXT
  FOR i=15 TO 11 STEP-1
    x=x+1
    CALL SetAPen&(rp&,i+mo)
    CALL Move&(rp&,x,y1)
    CALL Draw&(rp&,x,y2)
  NEXT
NEXT

a=-1
WHILE a
  me&=PEEKL(win&+86)
  abfrage: mess&=GetMsg&(me&):IF mess&=Ø THEN abfrage
  inm&=PEEKL(mess&+2Ø)     'Class of IntuitionMessage
  CALL ReplyMsg&(mess&)
  IF inm&=8 THEN a=Ø       'MOUSEBUTTONS = 8
WEND

ende:
IF win& THEN CALL CloseWindow&(win&)
IF scr& THEN CALL CloseScreen&(scr&)
IF rk& THEN CALL FreeRemember&(rk&,-1)
IF fehl THEN
  ON fehl GOSUB F1,F2,F3
  BEEP:PRINT ft-
END IF
LIBRARY CLOSE :END

F1:ft-= "Fehler bei OpenWindow":RETURN
F2:ft-= "Speicher nicht ausreichend":RETURN
F3:ft-= "FEHLER bei OpenScreen":RETURN

LibraryOeffnen:
### bitte hier einfügen ###
RETURN

Speicher:
### bitte hier einfügen ###
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
### bitte hier einfügen ###
END SUB
```

```
SUB Fenster (wb%,wh%,idf&,fl&,wtit&,tp%) STATIC
### bitte hier einfügen ###
END SUB
```

Der Anfang des Programmes bringt nichts Neues. Kommen wir daher gleich auf den Kern der Sache zu sprechen. In der äußeren FOR/NEXT-Schleife wird durch Zufallszahlen ermittelt, ob der rote, grüne oder blaue Farbanteil modifiziert werden soll. Die Variable *mo* erhält den Wert der entsprechenden Bit-Kombination der Planes 5 und 6. Nun wird eine schwarze Linie gezeichnet, die als Basis für die Modifikation dienen soll. Die erste innere Schleife zeichnet nur weitere senkrechte Linien daneben. Die Farben der Linien werden durch Modifikation mit den Bit-Werten 11 bis 15 immer heller. Die zweite innere Schleife modifiziert die Farben absteigend. Die folgenden Linien erhalten daher einen immer dunkler werdenden Farbton. Damit ist der 3-D-Effekt perfekt. Einfacher geht's wirklich nicht mehr.

Sicherlich haben Sie durch diese fünf Programm-Beispiele genug Anregungen für eigene Experimente oder Programme im HAM-Modus erhalten. Durch die Vielzahl an darstellbaren Farben steckt sicherlich noch einiges mehr in diesem Spezial-Modus des Amiga.

Kapitel 13

ExtraHalfBrite

Den View-Modus HAM mit der Darstellungsmöglichkeit von 4096 verschiedenen Farben kennt fast jeder Amiga-Besitzer. Der Modus *ExtraHalfBrite* dagegen, den Sie in diesem Kapitel kennenlernen werden, ist vielen unbekannt. Sie werden gleich feststellen, daß das Schattendasein dieses Modus völlig unbegründet ist. Immerhin können Sie damit 64 Farben auf den Bildschirm bringen!

13.1 64 Farben

Bevor wir diesen neuen Modus unter die Lupe nehmen, überlegen wir uns, wie der Amiga 64 Farben darstellen kann. Die Anzahl von 6 Bit-Ebenen, die der Amiga verwalten kann, würden 64 Farben zulassen. Die bekannte Formel

```
2 ^ Anzahl der BitPlanes = mögliche Farben
```

ergibt `2 ^ 6 = 64`

Farben. Mit 6 Bit-Ebenen lassen sich also theoretisch 64 Farben darstellen. Aber warum nur theoretisch? Das Problem liegt auch nicht bei der Software, sondern bei der Hardware des Amiga. Die Hardware des Amiga hat nämlich nur 32 Farbkontrollregister.

Auch im Modus *ExtraHalfBrite* wird mit 6 BitPlanes gearbeitet. Um aber, trotz der Einschränkungen bei den Farbkontrollregistern, dennoch 64 Farben auf den Bildschirm bringen zu können, werden die 32 Standard-Farben einmal normal und einmal modifiziert ausgegeben. Die Modifizierung stellt die Farben in ihrer halben Intensität dar. Damit haben Sie die normalen 32 Farben und 32 Farben mit halben Farbwerten, also insgesamt 64 zur Verfügung. Sie haben sicherlich bereits ein kleines Manko dieses View-Modus erkannt. Wird eine dunkle Farbe oder gar die Farbe Schwarz modifiziert, so ist der modifizierte Farbwert bei halber Intensität natürlich immer noch schwarz. Diesen vermeintlichen Nachteil kann man natürlich leicht umgehen. Man setzt einfach eine Farbtafel mit hellen Farben. Die dunklen Farbtöne erhält man dann durch die Modifikation. Das Bit der 6. BitPlane wird nur dazu gebraucht, um festzulegen, ob modifiziert werden muß bzw. welche Farbe genommen wird. Ist das Bit nicht gesetzt,

wird eine der ersten 32 Farben eingesetzt. Wenn das Bit gesetzt ist, wird eine der ersten 32 Farben modifiziert ausgegeben. Dabei werden die Standard-Farben wie üblich von den Farb-Nummern 0 bis 31 repräsentiert, während die modifizierten Farben durch die Farb-Nummern 32 bis 63 aufgerufen werden.

In der Maschinensprache wird eine Division durch 2 durch eine Rechtsverschiebung der Bits durchgeführt. Nach diesem Prinzip werden auch die Farben modifiziert. Schauen wir uns das Ganze im Zusammenhang an. Wir wollen im ExtraHalfBrite-Modus die Farb-Nummer 35 ausgeben. Damit ist das Bit der Bit-Ebene 6 gesetzt und die Farbwerte werden wie folgt modifiziert:

	Rot-Wert	**Grün-Wert**	**Blau-Wert**	
Farbe 3	1 1 1 1	1 0 0 0	0 0 0 0	orange
Farbe 35	0 1 1 1	0 1 0 0	0 0 0 0	braun

Durch die Rechtsverschiebung der Bits wird aus der Farbe Orange (Farbnummer 3) eine braune Farbe (Farbnummer 3+32=35). Durch die Halbierung der einzelnen Farbbestandteile erhalten wir eine dunklere Farbe als die entsprechende Original-Farbe.

Schauen Sie sich zum Schluß die ersten 11 Standard-Farben des Amiga als normale Farben und im HalfBrite-Modus an. Die Standard-Farben können Sie sich mit der Funktion *GetRGB4* der Grafik-Bibliothek aus der ColorMap holen.

Standard-Farben					**ExtraHalfBrite-Farben**				
Nr.	R	G	B	Farbe	Nr.	R	G	B	Farbe
0	0	5	A	blau	32	0	2	5	schwarz
1	F	F	F	weiß	33	7	7	7	grau
2	0	0	2	schwarz	34	0	0	1	schwarz
3	F	8	0	orange	35	7	4	0	braun
4	0	0	F	blau	36	0	0	7	dunkelblau
5	F	0	F	lila	37	7	0	7	dunkellila
6	0	F	F	hellgrün	38	0	7	7	dunkelgrün
7	F	F	F	weiß	39	7	7	7	grau
8	6	2	0	braunrot	40	3	1	0	schwarz
9	E	5	0	rot	41	7	2	0	dunkelrot
10	9	F	1	hellgrün	42	4	7	0	dunkelgrün

Sie sehen an dieser kleinen Tabelle, wie die Farben mit niedrigen Farbwerten zu schwarzen Farben werden. Eine eigene Farbtafel mit hellen Farbtönen, das bedeutet hohe Farbwerte, ist also durchaus empfehlenswert.

13.2 ExtraHalfBrite aktivieren

Um zu ergründen, welche Programm-Werkzeuge wir zur Programmierung des ExtraHalfBrite-Modus benötigen, fassen wir zusammen, was wir inzwischen über diesen Spezial-Modus alles wissen:

View-Modus EXTRA-HALFBRITE-Flag: 128
Anzahl der Bit-Ebenen: 6
Bildschirm-Breite: 320
Anzahl der Farben: 64

Eine Darstellung des HalfBrite-Modus ist mit den Befehlen des Standard-Basic nicht möglich, da die SCREEN-Anweisung nur maximal 5 BitPlanes zuläßt. Wir lassen uns daher in der bekannten Art und Weise von der Intuition einen neuen Bildschirm einrichten. Um die für Ihren Amiga mögliche Screenhöhe zu ermitteln, fragen wir nach der Höhe des Workbench-Screen. Die Bildschirm-Breite und die Tiefe des Screens kennen Sie bereits von der Aufstellung. Das Flag für den View-Modus (128) dürfen wir auf keinen Fall vergessen. Im Feld für die Type des Screens wird das Flag für CUSTOMSCREEN (15) gesetzt.

Aus den zurückliegenden Beispielen wissen Sie, daß es besser ist, zusätzlich ein Window einzurichten, damit über den IDCMP mit dem Programm korrespondiert werden kann. Aus einer NewWindow-Struktur macht uns die Intuition wieder ein Fenster. Das Feld für den IDCMP versorgen wir mit dem vom Programm abhängigen Flag. Das Typen-Feld erhält wie der Screen das Flag für CUSTOMSCREEN gesetzt.

Zur besseren Darstellung der Farben könnte man auch noch eine eigene Farbtafel mit Hilfe der Grafik-Routine *LoadRGB4* einlesen. Damit haben wir bereits alles zusammengestellt, was wir zur Programmierung des Extra-HalfBrite-Modus benötigen. Probieren wir das Wissen gleich an einem kleinen Beispiel aus.

13.3 SHADOW

Eine äußerst effektvolle Darstellung des ExtraHalfBrite-Modus ist die farbliche Darstellung von Licht und Schatten an einem Gegenstand. In dem folgenden Programmbeispiel wurde dieser Effekt eingesetzt. Eine Flotte von 32 verschiedenfarbigen Flugzeugen wird scheinbar von der linken Seite her von der Sonne bestrahlt. Dies wird dadurch erreicht, daß die linken Flugzeughälften in den 32 Standardfarben gefüllt werden und die rechten Hälften mit der zugehörigen Farbe im ExtraHalfBrite-Modus.

```
REM shadow  Pfad: Modus/13EHB/shadow
'P13-1
CLEAR
DEFINT a-z:DIM Flug1(2Ø),Flug2(2Ø),Flug11(2Ø),Flug22(2Ø)
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
sb=32Ø:tiefe=6:tp=15:stit&=Ø
vm=128                  'ViewModus  EXTRA HALFBRITE
wb=sb:wh=sh:idf&=8:fl&=4Ø96
t2-="Ich zeichne im Modus EXTRAHALFBRITE"+CHR-(Ø):wt2&=SADD(t2-)
t1-="SHADOW         linke Maustaste=ENDE"+CHR-(Ø):wt1&=SADD(t1-)
GOSUB LibraryOeffnen
GOSUB Speicher      :IF Fehl THEN ende
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,mb&) :IF Fehl THEN ende
CALL Fenster (wb,wh,idf&,fl&,wt2&,tp) :IF Fehl THEN ende
GOSUB InitTemp      :IF Fehl THEN ende
GOSUB zeichnen
CALL SetWindowTitles&(win&,wt1&,-1)

a=-1
WHILE a
  me&=PEEKL(win&+86)
  abfrage: mess&=GetMsg&(me&):IF mess&=Ø THEN abfrage
  cla&=PEEKL(mess&+2Ø)    'Class of IntuitionMessage
  CALL ReplyMsg&(mess&)
  IF cla&=8 THEN a=Ø
WEND

ende:
  IF win& THEN CALL CloseWindow&(win&)
  IF scr& THEN CALL CloseScreen&(scr&)
  IF bp& THEN CALL FreeRaster&(bp&,bmbr,bmRows)
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF Fehl THEN
    ON Fehl GOSUB f1,f2,F3
    BEEP:PRINT ft-
  END IF
  LIBRARY CLOSE :ERASE Flug1,Flug2,Flug11,Flug22
END

f1:ft-= "Fehler bei OpenWindow":RETURN
f2:ft-= "Speicher nicht ausreichend":RETURN
F3:ft-= "FEHLER bei OpenScreen":RETURN
```

```
LibraryOeffnen:
  DECLARE FUNCTION AllocRaster&() LIBRARY
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION OpenScreen&() LIBRARY
  DECLARE FUNCTION OpenWindow&() LIBRARY
  DECLARE FUNCTION GetMsg&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
  LIBRARY ":bue/exec.library"
RETURN

Speicher:
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  ms&=AllocRemember&(rk&,4Ø,art&) 'fuer NScreen-Struktur
  IF ms&=Ø THEN Fehl=2 :RETURN
  mw&=AllocRemember&(rk&,48,art&) 'fuer NWindow-Struktur
  IF mw&=Ø THEN Fehl=2 :RETURN
  mb&=AllocRemember&(rk&,2Ø,art&) 'fuer TmpRas-Struktur
  IF mb&=Ø THEN Fehl=2
RETURN

zeichnen:
  FOR i= Ø TO 19: READ Flug1(i):NEXT
  FOR i= Ø TO 19: READ Flug2(i):NEXT
  co=Ø :d1=Ø:d2=35:d3=(sh-2Ø)/8
  FOR b = Ø TO 3
    FOR h = Ø TO 7
      FOR i = Ø TO 19 STEP 2
        Flug11(i)=Flug1(i)+(b*7Ø)+3+d1
        Flug11(i+1)=Flug1(i+1)+(h*d3)+1Ø
      NEXT i
      CALL SetAPen&(wrp&,co)
      CALL move&(wrp&,Flug11(Ø),Flug11(1))
      CALL PolyDraw&(wrp&,1Ø,VARPTR(Flug11(Ø)))
      CALL Flood&(wrp&,1,Flug11(2)-1,Flug11(3)+1)
      FOR i = Ø TO 19 STEP 2
        Flug22(i)=Flug2(i)+(b*7Ø)+3+d1
        Flug22(i+1)=Flug2(i+1)+(h*d3)+1Ø
      NEXT i
      CALL SetAPen&(wrp&,co+32)
      CALL move&(wrp&,Flug22(Ø),Flug22(1))
```

```
      CALL PolyDraw&(wrp&,1Ø,VARPTR(Flug22(Ø)))
      CALL Flood&(wrp&,1,Flug22(2)+1,Flug22(3)+1)
      co=co+1 :SWAP d1,d2
  NEXT h,b
RETURN

InitTemp:
  bmbr=sb
  bmBytPerRow=bmbr/8      'Bytes per Grafikzeile
  bmRows=sh               'Grafik-Zeilen
  volum&=bmBytPerRow*bmRows
  bp&=AllocRaster&(bmbr,bmRows)
  IF bp&=Ø THEN Fehl=2:RETURN
  CALL BltClear&(bp&,volum&,Ø)
  'Fuer Struktur TmpRas
  POKEL mb&,bp&
  POKEL mb&+4,volum&
  CALL InitTmpRas&(mb&,bp&,volum&)
  POKEL wrp&+12,mb&
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
  SHARED ms&,scr&,rp&,vp&,Fehl%,ri&
  StrukturNeuerScreen:
  POKEW ms&,Ø      :POKEW ms&+2,Ø     'linke u.obere Kante
  POKEW ms&+4,sb% :POKEW ms&+6,sh%    'Breite, Hoehe
  POKEW ms&+8,tie%                    'Tiefe
  POKE ms&+1Ø,Ø    :POKE ms&+11,1     'DetailPen, BlockPen
  POKEW ms&+12,vm%                    'ViewModes EXTRAHALFBRITE
  POKEW ms&+14,tp%                    'Type  CUSTOMSCREEN
  POKEL ms&+16,Ø                      'TextAttr
  POKEL ms&+2Ø,stit&                  'Screen-Titel
  POKEL ms&+24,Ø  :POKEL ms&+28,sbm& 'Gadget,Bitmap
  scr&= OpenScreen&(ms&)
  IF scr&=Ø THEN Fehl%=3
  rp&=scr&+84 :vp&=scr&+44:ri&=PEEKL(scr&+8Ø)
END SUB

SUB Fenster (wb%,wh%,idf&,fl&,wtit&,tp%) STATIC
  SHARED mw&,scr&,win&,Fehl%,wrp&
  StrukturNeuesFenster:
  POKEW mw&,Ø      :POKEW mw&+2,Ø    'linke u.obere Kante
```

```
  POKEW mw&+4,wb% :POKEW mw&+6,wh%  'Breite, Hoehe
  POKE mw&+8,1    :POKE mw&+9,Ø     'DetailPen, BlockPen
  POKEL mw&+1Ø,idf&                 'IDCMPFlag MOUSEBUTTONS
  POKEL mw&+14,fl&                  'Flags ACTIVATE
  POKEL mw&+26,wtit&                'Window-Titel
  POKEL mw&+3Ø,scr&                 'Screen
  POKEW mw&+38,1ØØ:POKEW mw&+4Ø,3Ø  'min. Breite und Hoehe
  POKEW mw&+42,wb%:POKEW mw&+44,wh% 'max. Breite und Hoehe
  POKEW mw&+46,tp%                  'Type  CUSTOMSCREEN
  win&= OpenWindow&(mw&) :wrp&=PEEKL(win&+5Ø)
  IF win&=Ø THEN Fehl%=1
END SUB

DATA 18,Ø,24,Ø,24,32,22,3Ø,22,24
DATA Ø,Ø,4,Ø,22,18,22,4,18,Ø
DATA 31,Ø,25,Ø,25,32,27,3Ø,27,24
DATA 49,Ø,45,Ø,27,18,27,4,31,Ø
```

In der Subroutine »LibraryOeffnen« werden die Grafik-, die Intuition- und die Exec-Library geöffnet und die benötigten Funktionen als vom Basic heraus aufrufbar deklariert. Vom Speicherkuchen holen wir uns ein paar Scheiben für die Strukturen *NewScreen, NewWindow* und *TmpRas.*

Der Screen wird mit den Werten für die Screen-Breite *sb* von 320, mit einer *tiefe* von 6, dem View-Modus *vm* ExtraHalfBrite von 128 und dem Typ *tp 15* für CUSTOMSCREEN aufgerufen. Außerdem erhält er noch die Startadresse einer Bitmap, die zum Füllen der Flächen benötigt wird. Bei dem Aufruf der Subroutine »Fenster« sind die Variablen *idf&* für das IDCMP-Flag 8 (MOUSEBUTTONS) und *tp 15* für einen CUSTOMSCREEN wichtig.

Nachdem der temporäre Rastport eingerichtet ist, kann schon mit dem Zeichnen begonnen werden. Dazu werden die Daten-Werte in zwei Variablen-Felder eingelesen. Das Zeichnen der Grafiken selbst erfolgt in zwei ineinandergeschachtelten Schleifen mit den Grafik-Routinen *Move* und *PolyDraw.* Die linke Hälfte der Flugzeuge wird in den Standard-Farben mit *Flood* gefüllt. Die rechte Hälfte dagegen wird mit der modifizierten Farbe des ExtraHalfBrite-Modus ausgemalt.

Anschließend wartet das Programm in der WHILE/WEND-Schleife auf eine Betätigung der Maustaste. Mit den üblichen und notwendigen Aufräumungsarbeiten wird das Programm beendet.

Haben Sie das Programm schon laufen lassen? Das Spiel von Licht und Schatten ist doch beeindruckend, obwohl nur die Standard-Farben des Amiga eingesetzt wurden. Probieren Sie ruhig einmal das Programm mit einer eigenen Farbtafel aus. Sie werden sehen, daß sich das positive Ergebnis noch verbessern läßt.

13.4 Demonstration

Neben dem gerade gesehenen Einsatz für Schatteneffekte gibt es noch einige andere Möglichkeiten. Der View-Modus *ExtraHalfBrite* läßt sich auch hervorragend für Begrenzungslinien von gefüllten Flächen verwenden. Wenn Sie eine Grafik auf einem dunklen Untergrund zeichnen, kann für die äußeren Linien auch der Modus *ExtraHalfBrite* eingesetzt werden. Die Übergänge erscheinen dadurch weich und fließend. Das Wichtigste hätte ich bald vergessen: Natürlich stehen Ihnen 32 zusätzliche Farben zur Verfügung.

Das folgende Programm EXTRAHALFBRITE-DEMO zeigt Ihnen einige der besprochenen Beispiele dieses Spezial-Modus:

```
REM EXTRAHALFBRITE-DEMO  Pfad: Modus/13EHB/EHBdemo
'P13-2
CLEAR
DEFINT a-z:DIM Py(31),Py2(31)
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
sb=320:tiefe=6:tp=15:stit&=0
vm=128                    'ViewModus  EXTRA HALFBRITE
wb=sb:wh=sh:idf&=8:fl&=4096
GOSUB Texte
GOSUB LibraryOeffnen
GOSUB Speicher      :IF Fehl THEN ende
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,mb&) :IF Fehl THEN ende
CALL Fenster (wb,wh,idf&,fl&,wtl&,tp) :IF Fehl THEN ende
GOSUB InitTemp      :IF Fehl THEN ende
GOSUB zeichnen
CALL SetWindowTitles&(win&,wt5&,-1)
Taste wt5&

ende:
  IF win& THEN CALL CloseWindow&(win&)
  IF scr& THEN CALL CloseScreen&(scr&)
  IF bp& THEN CALL FreeRaster&(bp&,bmbr,bmRows)
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF Fehl THEN
    ON Fehl GOSUB f1,f2,F3
    BEEP:PRINT ft-
  END IF
  LIBRARY CLOSE :ERASE Py,Py2
END
```

```
f1:ft-= "Fehler bei OpenWindow":RETURN
f2:ft-= "Speicher nicht ausreichend":RETURN
F3:ft-= "FEHLER bei OpenScreen":RETURN

Texte:
t1-="EXTRAHALFBRITE  Maustaste=START"+CHR-(Ø):wt1&=SADD(t1-)
t2-="64 Farben       Maustaste=weiter"+CHR-(Ø):wt2&=SADD(t2-)
t3-="Schattenfuge    Maustaste=weiter"+CHR-(Ø):wt3&=SADD(t3-)
t4-="Kontur          Maustaste=weiter"+CHR-(Ø):wt4&=SADD(t4-)
t5-="EXTRAHALFBRITE  Maustaste=ENDE"+CHR-(Ø):wt5&=SADD(t5-)
RETURN

LibraryOeffnen:
  DECLARE FUNCTION AllocRaster&() LIBRARY
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION OpenScreen&() LIBRARY
  DECLARE FUNCTION OpenWindow&() LIBRARY
  DECLARE FUNCTION GetMsg&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
  LIBRARY ":bue/exec.library"
RETURN

Speicher:
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  ms&=AllocRemember&(rk&,4Ø,art&) 'fuer NScreen-Struktur
  IF ms&=Ø THEN Fehl=2 :RETURN
  mw&=AllocRemember&(rk&,48,art&) 'fuer NWindow-Struktur
  IF mw&=Ø THEN Fehl=2 :RETURN
  mb&=AllocRemember&(rk&,2Ø,art&) 'fuer TmpRas-Struktur
  IF mb&=Ø THEN Fehl=2
RETURN

zeichnen:
  CALL SetRGB4&(vp&,Ø,Ø,Ø,Ø)
  x=sb/2:y=sh/2
Rechteck:
  FOR i=1 TO 3
    Taste wt2&
    FOR fa= 1 TO 32
      d=33*i-fa*i:d1=d-1
      CALL SetAPen&(wrp&,fa+32)
      CALL RectFill&(wrp&,x-d,y-d,x+d,y+d)
```

```
      CALL SetAPen&(wrp&,fa)
      CALL RectFill&(wrp&,x-d1,y-d1,x+d1,y+d1)
    NEXT fa
  NEXT i
Fuge:
FOR fa = 3 TO 9 STEP 3
  Taste wt3&
  d3=5:d4=2
  FOR l = Ø TO 2
    x1=x-15Ø+(l*1ØØ):y1=y+4Ø:f1=fa+l:f2=fa+l+32
    FOR j=Ø TO 1Ø
      FOR i=y1 TO y1-d3 STEP -1
        CALL SetAPen&(wrp&,f1)
        CALL move&(wrp&,x1,i)
        CALL draw&(wrp&,x1+6Ø,i)
        CALL draw&(wrp&,x1+9Ø,i-3Ø)
        CALL draw&(wrp&,x1+3Ø,i-3Ø)
        CALL draw&(wrp&,x1,i)
        CALL Flood&(wrp&,1,x1+3,i-1)
      NEXT i
      y1=y1-d3:SWAP d3,d4:SWAP f1,f2
    NEXT j
  NEXT l
NEXT fa
Kontur:
  FOR i=Ø TO 31:READ Py(i):NEXT
  FOR i=Ø TO 31 STEP 2
    Py2(i)= Py(i)+1ØØ:Py2(i+1)=Py(i+1)
  NEXT
  FOR fa = 3 TO 14
    Taste wt4&
    CALL SetAPen&(wrp&,fa)
    CALL move&(wrp&,12Ø,16Ø)
    CALL PolyDraw&(wrp&,16,VARPTR(Py(Ø)))
    CALL SetAPen&(wrp&,fa+32)
    CALL move&(wrp&,22Ø,16Ø)
    CALL PolyDraw&(wrp&,16,VARPTR(Py2(Ø)))
    CALL SetAPen&(wrp&,fa)
    FOR d1=Ø TO 1ØØ STEP 1ØØ
      y1=62
      FOR x1=61 TO 14Ø STEP 25
```

```
        CALL Flood&(wrp&,1,x1+d1,y1):y1=y1+2Ø
      NEXT x1
    NEXT d1
  NEXT fa
Kreis:
  FOR i=1 TO 3
    Taste wt2&
    FOR fa= 1 TO 32
      r=33*i-fa*i
      CALL SetAPen&(wrp&,fa+32)
      CALL DrawEllipse&(wrp&,x,y,r,r)
      CALL SetAPen&(wrp&,fa)
      CALL Flood&(wrp&,1,x,y)
    NEXT fa
  NEXT i
RETURN

InitTemp:
  bmbr=sb
  bmBytPerRow=bmbr/8        'Bytes per Grafikzeile
  bmRows=sh                 'Grafik-Zeilen
  volum&=bmBytPerRow*bmRows
  bp&=AllocRaster&(bmbr,bmRows)
  IF bp&=Ø THEN Fehl=2:RETURN
  CALL BltClear&(bp&,volum&,Ø)
  'Fuer Struktur TmpRas
  POKEL mb&,bp&
  POKEL mb&+4,volum&
  CALL InitTmpRas&(mb&,bp&,volum&)
  POKEL wrp&+12,mb&
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
### bitte hier einfügen ###
END SUB

SUB Fenster (wb%,wh%,idf&,fl&,wtit&,tp%) STATIC
### bitte hier einfügen ###
END SUB

SUB Taste(tit&) STATIC
  SHARED win&,wrp&
  a=-1
```

```
  WHILE a
    me&=PEEKL(win&+86)
    abfrage: mess&=GetMsg&(me&):IF mess&=Ø THEN abfrage
    cla&=PEEKL(mess&+2Ø)    'Class of IntuitionMessage
    cod%=PEEKW(mess&+24)    'Code of IntuitionMessage
    CALL ReplyMsg&(mess&)
    IF cla&=8 AND cod%=1Ø4 THEN a=Ø
  WEND
  CALL SetRast&(wrp&,Ø)
  CALL SetWindowTitles&(win&,tit&,-1)
END SUB

Pyramide:
DATA 12Ø,16Ø,15Ø,13Ø,15Ø,6Ø,12Ø,9Ø,12Ø,16Ø
DATA 9Ø,16Ø,9Ø,9Ø,6Ø,6Ø,6Ø,13Ø,9Ø,16Ø
DATA 9Ø,9Ø,12Ø,9Ø,15Ø,6Ø,12Ø,3Ø,9Ø,3Ø
DATA 6Ø,6Ø
```

Zu Programm-Beginn gibt es gegenüber dem Vorprogramm nichts wesentlich Neues zu vermelden. Die beiden Unterprogramme »Schirm« und »Fenster« sind nicht mehr mit aufgeführt. Diese entnehmen Sie bitte dem Vorprogramm. – Das Haupt-Programm läuft in der Subroutine »zeichnen« ab. Bevor wir uns dort etwas näher umsehen, werfen wir einen Blick auf das Unterprogramm »Taste«. Die Adressen der Window-Struktur *win&* und der Rastport-Struktur des Fensters *wrp&* werden mit SHARED zu globalen Variablen. In der Schleife wird der Message-Port abgefragt. Wir fragen nach *Class* und *Code* der Message. Wenn *Class* = 8 (MOUSEBUTTONS) und *Code* = 104 (SELECT-DOWN), wird die Schleife verlassen. Die zusätzliche Abfrage von *Code* ist notwendig, da auch das Loslassen der Taste eine Message erzeugt. Würde also wie im letzten Programm nur *Class* gefragt, dann erhalten wir eine Meldung zuviel, bzw. das Unterprogramm müßte immer zweimal abgefragt werden. Nach dem Verlassen der Schleife wird durch *SetRast* mit der Hintergrundfarbe der Screen gelöscht (einschließlich Fenster-Titel). Anschließend setzen wir den neuen Titel des Fensters mit der übergebenen Adresse.

Nun kommen wir zur Subroutine »zeichnen«. Mit *SetRGB4* verpassen wir zuerst dem Hintergrund ein schwarzes Kleid. Zur Demonstration der 64 Farben werden mit *RectFill* drei verschiedene Größen mit jeweils 64 ineinandergelegten Rechtecken gezeichnet. Bei jedem äußeren Schleifendurchlauf wird das Unterprogramm »Taste« aufgerufen. Damit kann der Anwender das aktuelle Bild löschen und die nächste Grafik aufrufen. Durch den übergebenen Parameter zeigt die Titel-Leiste des Fensters einen Kurzkommentar zur jeweiligen Grafik.

Beim Label »Fuge« wird *ExtraHalfBrite* als Schattenfuge gezeigt. Es werden auf jedem Bild drei verschiedenfarbige Würfel gezeichnet. Durch das Umschalten von der Standardfarbe zur modifizierten Farbe sehen die Würfel wie gefüllte Bonbons aus. Die nächsten Grafiken demonstrieren den Modus als Kontur für eine Pyramide. Zur Verdeutlichung wird daneben die sechseckige Pyramide mit der Standard-Farbe gezeichnet. Gleichzeitig kann die Wirkung der modifizierten Farbe als Außenlinie getestet werden. Die Daten der Pyramide werden aus einem Datenfeld eingelesen und mit *PolyDraw* gezeichnet. Zum Schluß folgen, um das Programm abzurunden, 64 ineinandergezeichnete Kreisflächen.

Nachdem Sie nun einige Möglichkeiten des View-Modus *ExtraHalfBrite* kennengelernt haben, werden Sie mir sicher zustimmen, daß der Modus kein Schattendasein unter den anderen Modi zu führen braucht.

Kapitel 14

Dual PlayField

Jetzt kommen wir zum interessantesten View-Modus des Amiga. Im Zusammenhang mit Basic haben Sie den Modus *Dual PlayField* wahrscheinlich noch nicht gehört. Dabei lassen sich die tollsten Dinge damit anstellen. Wie schon der Name sagt, können Sie damit für einen Bildschirm zwei Grafik-Ebenen, sogenannte PlayFields, programmieren. Diese lassen sich dann einzeln oder kombiniert ausgeben.

Das klingt ja schier unglaublich. Vielleicht ahnen Sie bereits, was Sie damit alles anfangen können. Da läuft Ihnen wahrscheinlich schon das Wasser im Munde zusammen und Ihre Hände lauern schon in Programmierstellung auf den Startschuß. Endlich können Sie das Spiel programmieren, dessen Bildvordergrund die Inneneinrichtung eines Panzers und dessen Hintergrund eine schier unendliche Landschaft zeigt, die sich unabhängig vom Vordergrund vorbeiscrollen läßt. Oder der Vordergrund ist das Innere eines Raumschiffes und der bewegliche Hintergrund der Weltraum. Oder der Vordergrund ist eine Grafik und als Hintergrund rollt eine überdimensionale Schrift vorbei. Oder oder oder ...

Bevor wir weiter darin schwelgen, was damit alles realisiert werden kann, kommen wir lieber wieder auf den Boden der rauhen Wirklichkeit zurück. Nicht umsonst haben Sie von diesem Modus bisher recht wenig im Zusammenhang mit dem Amiga-Basic gehört. Wie bei alle schönen Sachen ist es auch hier so, daß es ohne Fleiß keinen Preis gibt. Dieser Modus ist also etwas schwieriger zu programmieren als die bisher besprochenen Modi. Aber so schwer zu programmieren ist er auch wieder nicht.

14.1 Aus eins mach zwei oder umgekehrt

Genau genommen haben wir es beim Dual PlayField sogar mit zwei View-Modi zu tun. Der zweite Modus macht aber keine zusätzliche Arbeit. Im Gegenteil, er erhöht zusätzlich die Vielseitigkeit des Modus:

View-Modus	hex (Bit)	dezimal	Kurzbeschreibung
PFBA	$40 (6)	64	PlayField-Priorität für DUALPF
DUALPF	$400 (10)	1024	zwei PlayFields

Mit dem Bit 10 im Modus-Feld der ViewPort-Struktur wird der Modus DUALPF (Dual PlayField) aktiviert. Wir können damit, wie bereits erwähnt, zwei PlayFields oder Raster darstellen. Die beiden Bitmaps müssen nicht die gleiche Größe haben. Da die Kontrolle darüber unabhängig voneinander erfolgt, sind Ihren Wünschen nur insoweit Grenzen gesetzt, wie Grafik-Speicher zur Verfügung steht. Normalerweise liegt das PlayField A oben bzw. im Bildvordergrund und das zweite Playfield B liegt darunter. Diese Reihenfolge können wir aber jederzeit ändern. Dazu brauchen wir nur zusätzlich das Bit 6 PFBA in den View-Modus schreiben. Die Abkürzung PFBA bedeutet wahrscheinlich PlayField B-A (B über A), zur Unterscheidung zur normalen Darstellung PlayField A-B (A über B).

Die Tiefe des Bildschirmes können Sie frei zwischen 2 und 6 bestimmen. Natürlich ergibt sich bei der Farbgebung eine Einschränkung. Die separat kontrollierten Raster können natürlich nicht gemeinsam zur Bit-Kombination für die Anzahl der Farben herangezogen werden. Jedes PlayField kann daher nur in sich die Bits für die einzelnen Farben kombinieren. Zusätzlich kann das unterste Bit nur bedingt zur Farbgebung eingesetzt werden. Die Hintergrundfarbe wird durch die Farbe 0 von PFA (PlayField A) gesetzt. Die Hintergrundfarbe ist ja bekanntermaßen auch für die Randfarbe des Video-Bildes verantwortlich. Wir werden später bei der Programmierung darauf zurückkommen. Ansonsten ist die Farbe 0 transparent. Schließlich muß es ja eine Möglichkeit geben, das darunterliegende PFB zu sehen.

Das unterste Bit von PFB wird immer durch die Farbnummer 8 repräsentiert. Diese Farbe ist immer gleichzeitig die Hintergrundfarbe 0 von PFA. Sie kann daher auch nicht durch die Routine *SetRGB4* verändert werden. Das ist auch logisch, denn beim Wechsel der Prioritäten wird diese Farbe transparent. Damit kann dann auf das darunterliegende PFA geschaut werden.

Für die Anzahl der Bit-Ebenen der beiden Raster der PlayFields stehen fünf verschiedene Kombinationen zur Auswahl. In der folgenden Tabelle sind sie mit den zugehörigen Farbnummern zusammengestellt:

Bit-Ebenen gesamt	**Bit-Ebenen PlayField A**	**Farb-Register**	**Bit-Ebenen PlayField B**	**Farb-Register**
2	1	0–1	1	8–9
3	2	0–3	1	8–9
4	2	0–3	2	8–11
5	3	0–7	2	8–11
6	3	0–7	3	8–15

Durch die Änderung des Flags PFBA für die Priorität ergeben sich noch weitere Kombinationen. So können Sie zum Beispiel mit 3 Bit-Ebenen auch PFA eine BitPlane und PFB zwei Bit-Ebenen zuweisen.

Sie haben gerade erfahren, daß wir es im Modus *Dual PlayField* mit zwei unabhängigen Rastern zu tun haben. Damit in diese Raster gezeichnet werden kann, benötigen wir für beide einen eigenen RastPort und eine eigene RasInfo-Struktur. Damit der Zusammenhang nicht verlorengeht, schauen Sie sich am besten Bild 14.1 über die Grafik-Ausgabeelemente beim Modus *Dual PlayField* an.

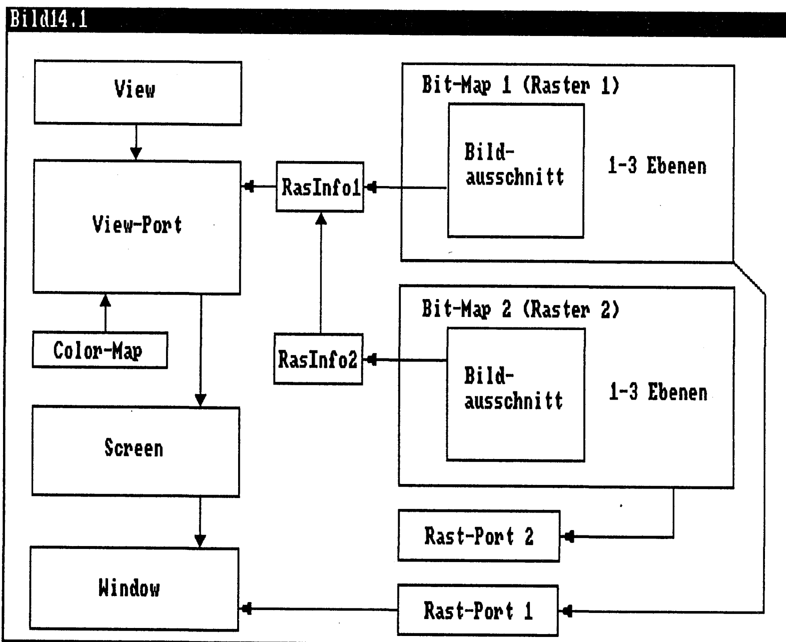


Bild 14.1: Die Grafik-Ausgabe-elemente beim Modus DUALPF

Die Schaltzentrale für die ganze Ausgabe ist der ViewPort. Da ein ViewPort nur eine ColorMap haben kann, sind die Farben der beiden PlayFields in einer Farbtabelle (siehe Übersicht) zu finden. Die andere Verbindung geht über die RasInfo-Struktur zur Bitmap. Da beim Modus DUALPF die Bitmap in zwei Teile zerlegt ist, müssen auch zwei RasInfo-Strukturen angelegt sein. Die beiden Strukturen sind durch einen Zeiger miteinander verknüpft. (Einzelheiten können Sie den Strukturen im Anhang entnehmen.)

Kommen wir nun zum Rastport, oder besser gesagt zu den beiden RastPorts. Wenn das Bild ohne Window erstellt wird, besteht keine direkte Verbindung zum Screen, sondern nur zur Bitmap. Das ist verständlich, denn eigentlich zeichnen wir nicht in den Screen, sondern in dessen Bitmap. Wenn also ein zweiter RastPort benötigt wird, genügt die Verbindung zur Bitmap. Damit kann dann bereits in den RastPort gezeichnet werden. Die Verbindung zum ViewPort läuft dann über die RasInfo-Struktur. Diese vereinfachte Darstellung soll genügen. Schließlich haben wir in der Intuition noch einen mächtigen Verbündeten, dem wir die Feinabstimmung überlassen können.

14.2 Von der Theorie zur Praxis

Bei der Programmierung des View-Modus *Dual PlayField* unter Basic können wir die Reihenfolge Bitmap-RasInfo-ViewPort-Screen nicht einhalten. Den neuen ViewPort erhalten wir ja bereits, wenn wir einen Screen öffnen. Suchen wir also einen Weg, den das System akzeptiert, ohne uns den großen Guru auf den Hals zu hetzen. Um nichts zu vergessen, fassen wir kurz zusammen, was wir alles benötigen:

ViewPort	durch Intuition
Screen	durch Intuition
Window	durch Intuition (nicht unbedingt erforderlich)
Bitmap1	durch Programm
Bitmap2	durch Programm
RasInfo1	durch Programm
RasInfo2	durch Programm
RastPort1	durch Intuition
RastPort2	durch Programm

Hoffentlich schmeißen Sie jetzt nicht das Handtuch. Das sieht gewaltiger aus als es ist.

Zuerst benötigen wir eine Anzahl von Speicherbereichen. Fangen wir mit den verschiedenen Strukturen an. Mit *AllocRemenber* reservieren wir System-Speicher für:

NewScreen-Struktur
NewWindow-Struktur
Bitmap-Struktur 1

Bitmap-Struktur 2
RasInfo-Struktur 1
RasInfo-Struktur 2
RastPort-Struktur 2

Als nächstes wagen wir uns an die großen Speicherbereiche. Mit der Grafik-Routine *AllocRaster* holen wir uns die Bereiche für die beiden Bitmaps. Hier heißt es, vorsichtig zu sein. Die Speicherkapazität der unteren 512 Kbyte, die für die Grafik-Programmierung zur Verfügung stehen, ist schnell erschöpft.

Bei dieser Gelegenheit können wir auch gleich die beiden Bitmap-Strukturen und die beiden RasInfo-Strukturen anlegen, da alle benötigten Parameter bekannt sind. Die RasInfo-Strukturen legen wir direkt hintereinander an, damit wir sie gleich durch einen Zeiger miteinander verknüpfen können. Nun können wir auch die NewScreen-Struktur erstellen. Neben den bekannten Parametern sind die folgende Werte für den Modus *Dual PlayField* von Interesse:

Feld der Struktur	**Bemerkung**
Depth	Tiefe der beiden Bitmaps zusammen
ViewModes	DUALPLAYFIELD, Wert 1024
Type	CUSTOMSCREEN + CUSTOMBITMAP, Wert 79
CustomBitmap	Zeiger auf die erste Bitmap (PlayField A)

Mit *OpenScreen* aktivieren wir gleich anschließend den neuen Screen. Nun kann auch das neue Window geöffnet werden. Aus der Fenster-Struktur kann dann die Adresse des ViewPorts geholt werden. Da der ViewPort nicht unseren Vorstellungen entspricht, werden wir ihn nun etwas zurechtbiegen. Damit es bei der Rückkehr aus den Modus DUALPF kein böses Erwachen gibt, retten wir die Adresse der alten RasInfo-Struktur in einer Variablen. Nun schreiben wir die Adresse unserer eigenen RasInfo-Struktur an diese Stelle. Mit der Routine *RemakeDisplay* bitten wir die Intuition um die Abstimmung mit dem Rest der Grafik-Welt.

Das war's schon. Halt, eines haben wir noch vergessen. Damit wir auch in beide Bitmaps zeichnen können, brauchen wir noch eine RastPort-Struktur für die zweite Bitmap. Dazu rufen wir die Routine *InitRastPort* aus der Grafik-Bibliothek auf. Dadurch erhalten wir einen neuen RastPort, der mit den Standard-Werten ausgerüstet ist, was für unsere Zwecke ausreicht. An die vierte Speicherstelle schreiben wir noch schnell die Adresse der zweiten Bitmap, und damit sind wir wirklich fertig. Nun kann das normale Programm mit Grafik erstellen etc. folgen.

Wenn Sie dann das Programm beenden, dürfen Sie nicht vergessen, die gerettete Adresse der alten RasInfo-Struktur in die entsprechende Speicherstelle des View-Ports zu schreiben. Das muß geschehen, bevor der Screen geschlossen und die Speicherbereiche freigegeben werden. Wenn der Amiga sich am Ende des Programmes sang- und klanglos verabschiedet, prüfen Sie zuerst, ob Sie diesen Punkt beachtet haben.

14.3 Blende auf und zu

Mit soviel Theorie vollgepfropft wagen wir uns zuerst an ein einfaches Beispiel heran. Sie kennen doch sicher vom Fernsehen die tollen Effekte, wo ein Bild aus einem anderen Bild entsteht oder von einem anderen Bild verschluckt wird. Unser erstes Programm im View-Modus *Dual PlayField* demonstriert mit einfachen Mitteln solche Effekte.

```
REM Blende  Pfad: Modus/14DUALPF/Blende
'P14-1
CLEAR
DEFINT a-z
Bildschirm:
  wbs&=PEEKL(WINDOW(7)+46)     'Workbench-Screen
  sb=320                       'Breite
  sh=PEEKW(wbs&+14)            'Hoehe
  tiefe=5                      'Tiefe
  vm=1024                      'View-Modus  DUALPLAYFIELD
  tp=79                        'Type CUSTOMSCREEN CUSTOMBITMAP
  stit&=0                      'Screen-Titel
  mb1&=0                       'CustomBitmap
Bitmap1:                       'Farbe 0-7
  bmbr1=sb+2                   'Breite
  bmBytPerRow1=bmbr1/8         'Bytes pro Zeile
  bmRows1=sh                   'Grafik-Zeilen
  tiefe1=3                     'Tiefe
  volum1&=bmBytPerRow1*bmRows1 'Groesse
Bitmap2:                       'Farbe 8-11
  bmbr2=sb                     'Breite
  bmBytPerRow2=bmbr2/8         'Bytes pro Zeile
  bmRows2=sh                   'Grafik-Zeilen
  tiefe2=2                     'Tiefe
  volum2&=bmBytPerRow2*bmRows2 'Groesse
```

```
GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
GOSUB Bitmapanlegen :IF fehl THEN ende
GOSUB RasInfoStruktur
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,mb1&) :IF fehl THEN ende
vp&=scr&+44
'alte RasInfo-Adresse retten
  rialt&=PEEKL(vp&+36)
'neue RasInfo-Adresse in ViewPort schreiben
  POKEL vp&+36,ri&
  CALL remakedisplay&
'RastPort fuer Bitmap2 anlegen
  CALL InitRastPort&(rp2&)
  POKEL rp2&+4,mb2&
GOSUB zeichnen

CALL SetAPen&(rp&,Ø)
FOR i = 1 TO 315
  CALL RectFill&(rp&,i,Ø,i,sh-1)
NEXT i
CALL SetAPen&(rp&,1)
FOR i = 315 TO Ø STEP -1
  CALL RectFill&(rp&,i,Ø,i,sh-1)
NEXT i
CALL SetAPen&(rp&,Ø)
FOR i = 1 TO sh
  CALL RectFill&(rp&,Ø,i,32Ø,i)
NEXT i
CALL SetAPen&(rp&,1)
FOR i = sh TO 1 STEP -1
  CALL RectFill&(rp&,Ø,i,32Ø,i)
NEXT i
xb=sb/2:yh=sh/2
CALL SetAPen&(rp&,Ø)
FOR i = 1 TO yh-2
  CALL RectFill&(rp&,xb-i,yh-i,xb+i,yh+i)
NEXT i
FOR i= 1 TO 1ØØØ:NEXT

  POKEL vp&+36,rialt&
  CALL remakedisplay&
```

```
ende:
  IF scr& THEN CALL CloseScreen&(scr&)
  FOR i = Ø TO tiefe1-1
    IF bp1&(i) THEN CALL FreeRaster&(bp1&(i),bmbr1,bmRows1)
  NEXT
  FOR i = Ø TO tiefe2-1
    IF bp2&(i) THEN CALL FreeRaster&(bp2&(i),bmbr2,bmRows2)
  NEXT
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN
    ON fehl GOSUB f1,f2
    BEEP:PRINT ft-
  END IF
  LIBRARY CLOSE
END

f2:ft-= "Speicher nicht ausreichend":RETURN
f1:ft-= "FEHLER bei OpenScreen":RETURN

RasInfoStruktur:
  POKEL ri&,ri&+12        'Next
  POKEL ri&+4,mb1&        '1.BitMap
  POKEW ri&+8,Ø           'RxOffset
  POKEW ri&+1Ø,Ø          'RyOffset
  POKEL ri&+12,Ø          'Next
  POKEL ri&+16,mb2&       '2.BitMap
  POKEW ri&+2Ø,Ø          'RxOffset
  POKEW ri&+22,Ø          'RyOffset
RETURN

LibraryOeffnen:
  DECLARE FUNCTION AllocRaster&() LIBRARY
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION OpenScreen&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
RETURN

Speicher:
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  ms&=AllocRemember&(rk&,4Ø,art&)    'fuer NScreen-Struktur
  IF ms&=Ø THEN fehl=2 :RETURN
  mb1&=AllocRemember&(rk&,4Ø,art&)   'fuer Bitmap-Struktur1
```

```
  IF mb1&=Ø THEN fehl=2 :RETURN
  mb2&=AllocRemember&(rk&,4Ø,art&)  'fuer Bitmap-Struktur2
  IF mb2&=Ø THEN fehl=2 :RETURN
  ri&=AllocRemember&(rk&,4Ø,art&)   'fuer RasInfoStruktur
  IF ri&=Ø THEN fehl=2 :RETURN
  rp2&=AllocRemember&(rk&,8Ø,art&)  'fuer RastPort2
  IF rp2&=Ø THEN fehl=2
RETURN

Bitmapanlegen:
  FOR i = Ø TO tiefe1-1
    bp1&(i)=Ø
    bp1&(i)=AllocRaster&(bmbr1,bmRows1)
    IF bp1&(i)=Ø THEN fehl=2:RETURN
    CALL BltClear&(bp1&(i),volum1&,Ø)
  NEXT
  FOR i = Ø TO tiefe2-1
    bp2&(i)=Ø
    bp2&(i)=AllocRaster&(bmbr2,bmRows2)
    IF bp2&(i)=Ø THEN fehl=2:RETURN
    CALL BltClear&(bp2&(i),volum2&,Ø)
  NEXT
  POKEW mb1&,bmBytPerRow1         'Bytes/Reihe
  POKEW mb1&+2,bmRows1            'Reihen
  POKE mb1&+5,tiefe1              'Tiefe
  FOR n = Ø TO tiefe1-1
    POKEL mb1&+8+(n*4),bp1&(n)    'n BitPlanes
  NEXT
  POKEW mb2&,bmBytPerRow2         'Bytes/Reihe
  POKEW mb2&+2,bmRows2            'Reihen
  POKE mb2&+5,tiefe2              'Tiefe
  FOR n = Ø TO tiefe2-1
    POKEL mb2&+8+(n*4),bp2&(n)    'n BitPlanes
  NEXT
RETURN

zeichnen:
  CALL SetRGB4&(vp&,Ø,12,12,12)     'hellgrau
  CALL SetRGB4&(vp&,1,12,12,12)     'hellgrau
  CALL SetRast&(rp2&,9)
  CALL SetRast&(rp&,1)
```

```
  FOR i = Ø TO 8Ø
    x=RND*sb-3Ø:y=RND*sh-3Ø:d=RND*25:fa=RND*3+8
    CALL SetAPen&(rp2&,fa)
    CALL RectFill&(rp2&,x,y,x+d,y+d)
  NEXT
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
  SHARED ms&,scr&,rp&,fehl%
  POKEW ms&,Ø      :POKEW ms&+2,Ø       'linke u.obere Kante
  POKEW ms&+4,sb% :POKEW ms&+6,sh%     'Breite, Hoehe
  POKEW ms&+8,tie%                     'Tiefe
  POKE ms&+1Ø,Ø    :POKE ms&+11,1       'DetailPen, BlockPen
  POKEW ms&+12,vm%:POKEW ms&+14,tp%    'ViewModes, Type
  POKEL ms&+16,Ø   :POKEL ms&+2Ø,stit& 'TextAttr, Screen-Titel
  POKEL ms&+24,Ø   :POKEL ms&+28,sbm&  'Gadget,Bitmap
  scr&= OpenScreen&(ms&) :IF scr&=Ø THEN fehl%=1
  rp&=scr&+84
END SUB
```

Gleich zu Beginn finden Sie eine Auflistung der Variablen für die Grafik-Ausgabe. Alle sind kommentiert, so daß sich eine weitere Erklärung erübrigt. Anschließend werden die Intuition- und die Grafik-Library geöffnet. Die folgende Reservierung der Speicherbereiche für die Strukturen haben wir bereits besprochen. Die beiden neuen Bitmaps haben die Größe des Screens. Nun sind die neuen RasInfo-Strukturen an der Reihe. Die erste Speicherstelle der ersten Struktur zeigt 12 Speicherstellen weiter, wo bereits die zweite RasInfo-Struktur beginnt. Jeweils an die vierte Speicherstelle werden die Adressen der beiden Bitmap-Strukturen geschrieben. Da das Bild nicht gescrollt wird, erhalten *Rx* und *Ry* die normalen Werte Null.

Nun kann bereits das Unterprogramm »Schirm« aufgerufen werden. Auch hier haben wir bereits über die wichtigen Parameter gesprochen. Mit der Routine *OpenScreen* wird die neue Screen-Struktur zusammen mit einem neuen ViewPort angelegt. Um das Programm möglichst kurz zu halten, verzichten wir auf eine Kommunikation zwischen Anwender und Programm. Wir benötigen daher auch kein Window. Die Window-Struktur würden wir aber benötigen, um mit der Library-Routine *ViewPortAddress* die Adresse des View-Ports zu ermitteln. Wir suchen und finden die ViewPort-Struktur aber auch innerhalb der neu angelegten Screen-Struktur ab der 44. Speicherstelle. Mit der Adresse der ViewPort-Struktur retten wir in der Variablen *rialt&* die alte RasInfo-Struktur. Mit dem gleichen Offset von 36 der ViewPort-Struktur schreiben wir unsere neue RasInfo-Struktur. Zum Schluß der Vorbereitungsphase öffnen wir den zweiten RastPort für die zweite Bitmap.

Die Subroutine »zeichnen« ist recht kurz gehalten. Zuerst werden die Hintergrundfarbe und die Farbe 1 des PlayFields A auf die gleiche Farbe gesetzt. Mit dieser Farbe wird der RastPort des PFA gefüllt. Um einen besseren Kontrast zu erhalten, wird der RastPort des unteren PFB mit der Farbe 9 (normalerweise rot) gefüllt. Nun werden noch ein paar Rechtecke in den Farben 8–11 gezeichnet, wovon die Rechtecke mit der Farbe 9 natürlich nicht zu sehen sind.

Die Subroutine »zeichnen«, so unscheinbar sie auch aussehen mag, zeigt eine wichtige Technik für den DUALPF-Modus. Damit die Tatsache einer verdeckten Ebene, auch für den unvoreingenommenen Betrachter, wirkungsvoll dargestellt werden kann, muß sie perfekt funktionieren. Das geht aber nicht, wenn an allen passenden und unpassenden Stellen des Screens das verdeckte PlayField hervorspitzt (Sie finden später ein solches Beispiel). Dazu wird zuerst die Randfarbe und die Hauptfarbe des sichtbaren PFA mit der gleichen Farbe versehen. Leider bleibt dann immer noch ein zwei Pixel breiter Streifen an der rechten Bildseite, durch den das untere PFB zu sehen ist. Wenn Sie die Variablen am Programmanfang genau betrachten, werden Sie feststellen, daß die Bitmap 1 für das PFA um 2 Pixel verbreitert wurde. Damit ist auch dieses Hindernis beseitigt.

Wenden wir uns kurz den einfachen Schleifen des Hauptprogrammes zu. Für das PlayField A setzen wir die Zeichenfarbe 0, das heißt transparent ein. Nun zeichnen wir in der FOR/NEXT-Schleife von links nach rechts lauter senkrechte Linien in der transparenten Farbe. Es tritt das untere PlayField zutage. Die nächste Schleife zeichnet in entgegengesetzter Richtung mit der Farbe 1. Der Vorhang vor dem unteren PlayField wird also schnell auf- und genauso schnell wieder zugezogen. Die anderen Schleifen sind ähnlich aufgebaut. Damit das erste Programm nicht zu lang wird, wurde auf eine eigene Grafik für das PlayField A verzichtet. Der optische Eindruck ist auch ohne zusätzliche Schnörkel recht passabel. Bevor die Aufräumungsarbeiten beginnen, folgen zwei sehr wichtige Programmzeilen. Die gerettete RasInfo-Struktur wird in den ViewPort geschrieben.

Sie sehen an dem Beispiel, daß es auch vom Basic heraus möglich ist, den Modus DUALPF zu programmieren. Das Ergebnis hat sich auf jeden Fall gelohnt. Und der Programmier-Aufwand hat sich auch in Grenzen gehalten. Der Aufwand ist jedenfalls geringer als in jeder anderen Programmiersprache.

14.4 Sea Patrol

Das erste Beispiel war zwar schon recht beeindruckend, aber die Feinheiten des Dual-PlayField-Modus kommen erst bei größeren Bitmaps zum Tragen. Dabei darf natürlich ein ordentliches PlayField-Scrolling nicht fehlen. (Solche frommen Wünsche schlagen sich natürlich in der Programmlänge nieder.) Da sich außerdem der Modus auch hervorragend für Spiele eignet, sollen Sie gleich an einem kleinen Spiel die Stärke von DUALPF demonstriert bekommen. Viel Spaß!

```
REM SeaPatrol  Pfad: Modus/14DUALPF/SeaPatrol
'P14-2
CLEAR
DEFINT a-z :DIM we(255)
GOSUB Inseln
Bildschirm:
  wbs&=PEEKL(WINDOW(7)+46)       'Workbench-Screen
  wrp&=WINDOW(8)                 'Window-Rastport
  sb=322                         'Breite
  sh=PEEKW(wbs&+14) 'Hoehe
  tiefe=4                        'Tiefe
  vm=1024                        'View-Modus  DUALPLAYFIELD
  tp=79                          'Type CUSTOMSCREEN CUSTOMBITMAP
  stit&=0                        'Screen-Titel
  mb1&=0                         'CustomBitmap
Bitmap1:                         'Farbe 0-3
  bmbr1=sb                       'Breite
  bmBytPerRow1=bmbr1/8           'Bytes pro Zeile
  bmRows1=sh                     'Grafik-Zeilen
  tiefe1=2                       'Tiefe
  volum1&=bmBytPerRow1*bmRows1 'Groesse 20480/16000 Byte
Bitmap2:                         'Farbe 8-11
  bmbr2=512                      'Breite
  bmBytPerRow2=bmbr2/8           'Bytes pro Zeile
  bmRows2=1024                   'Grafik-Zeilen
  tiefe2=2                       'Tiefe
  volum2&=bmBytPerRow2*bmRows2 'Groesse 131072 Byte

GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
GOSUB Bitmapanlegen :IF fehl THEN ende
GOSUB RasInfoStruktur
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,mb1&) :IF fehl THEN ende
vp&=scr&+44
'alte RasInfo-Adresse retten
  rialt&=PEEKL(vp&+36)
'neue RasInfo-Adresse in ViewPort schreiben
  POKEL vp&+36,ri&
  CALL remakedisplay&
'RastPort fuer Bitmap2 anlegen
  CALL InitRastPort&(rp2&)
```

```
  POKEL rp2&+4,mb2&
GOSUB zeichnen
FOR i=Ø TO 255:we(i)=RND*255:we(i)=we(i)-128:NEXT
WAVE 1,we:ERASE we

CALL SetBPen&(rp&,1)
txt-="Gas":te&=SADD(txt-)
CALL Move&(rp&,125,198):CALL Text&(rp&,te&,3)
a=-1:xp=Ø:yp=17Ø:x=Ø:y=Ø:pu=Ø:gas=3ØØØØ

WHILE a
  txt-="Punkte"+STR-(pu):lg=LEN(txt-):te&=SADD(txt-)
  CALL Move&(rp&,22Ø,198):CALL Text&(rp&,te&,lg)
  gas=gas-tempo:ga=gas/5ØØ
  CALL SetAPen&(rp&,1)
  CALL RectFill&(rp&,15Ø,194,15Ø+ga+1,199)
  CALL SetAPen&(rp&,3)
  CALL RectFill&(rp&,15Ø,194,15Ø+ga,199)
  IF STICK(3) <Ø THEN
    yp=yp-1: IF yp<15Ø THEN yp=15Ø
    tempo=tempo+1: IF tempo>8 THEN tempo=8
  ELSE
    yp=yp+1 :IF yp>17Ø THEN yp=17Ø
    tempo=tempo-1: IF tempo<1 THEN tempo=1
  END IF
  y=y-tempo:IF y<Ø THEN y=873
  IF STICK(2) THEN
    x=x+STICK(2)
    IF STICK(2)>Ø THEN xp=Ø ELSE xp=14Ø
  ELSE
    xp=7Ø
  END IF
  SOUND 1Ø+tempo*3,1.7,95+(tempo*2Ø),1
  IF x>511 THEN x=Ø
  IF x<Ø THEN x=x+511
  IF warte<Ø THEN
    IF y>72Ø AND y<74Ø THEN
      IF x>462 AND x<482 THEN Ereignis=-1:warte=2Ø
    END IF
    IF y>9Ø AND y<11Ø THEN
      IF x>3ØØ AND x<32Ø THEN Ereignis=-1:warte=2Ø
    END IF
```

```
  END IF
  IF Ereignis THEN BEEP:Ereignis=Ø:pu=pu+1
  warte =warte-1
  CALL ClipBlit&(wrp&,xp,Ø,rp&,5Ø,yp,7Ø,7Ø,192)
  POKEW ri&+2Ø,x
  POKEW ri&+22,y
  CALL RethinkDisplay&
  CALL ScrollVPort& (vp&)
  IF STRIG(3) THEN a=Ø
  IF gas<Ø THEN a=Ø
WEND

  POKEL vp&+36,rialt&
  CALL remakedisplay&

ende:
  CLS
  IF scr& THEN CALL CloseScreen&(scr&)
  FOR i = Ø TO tiefe1-1
    IF bp1&(i) THEN CALL FreeRaster&(bp1&(i),bmbr1,bmRows1)
  NEXT
  FOR i = Ø TO tiefe2-1
    IF bp2&(i) THEN CALL FreeRaster&(bp2&(i),bmbr2,bmRows2)
  NEXT
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN
    ON fehl GOSUB f1,f2
    BEEP:PRINT ft-
  END IF
  LIBRARY CLOSE
  PRINT "Ihre Punkte: "pu
END

f2:ft-= "Speicher nicht ausreichend":RETURN
f1:ft-= "FEHLER bei OpenScreen":RETURN

RasInfoStruktur:
  POKEL ri&,ri&+12       'Next
  POKEL ri&+4,mb1&       '1.BitMap
  POKEW ri&+8,Ø          'RxOffset
  POKEW ri&+1Ø,Ø         'RyOffset
  POKEL ri&+12,Ø         'Next
  POKEL ri&+16,mb2&      '2.Bitmap
```

```
  POKEW ri&+2Ø,Ø          'RxOffset
  POKEW ri&+22,Ø          'RyOffset
RETURN

LibraryOeffnen:
  DECLARE FUNCTION AllocRaster&() LIBRARY
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION OpenScreen&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
RETURN

Speicher:
### Bitte hier einfügen! ###
RETURN

Bitmapanlegen:
### Bitte hier einfügen ###
RETURN

zeichnen:
  'Farben
    CALL SetRGB4&(vp&,Ø,12,12,12)     'hellgrau
    CALL SetRGB4&(vp&,1,12,12,12)     'hellgrau
    CALL SetRGB4&(vp&,2,7,7,7)        'mittelgrau
    CALL SetRGB4&(vp&,3,Ø,Ø,Ø)        'schwarz
    CALL SetRGB4&(vp&,9,4,2,15)       'blau
    CALL SetRGB4&(vp&,1Ø,Ø,15,Ø)      'gruen
    CALL SetRGB4&(vp&,11,15,12,1Ø)    'sandfarben
    CALL SetRast&(rp2&,9)
    CALL SetRast&(rp&,1)
  'oberes Playfield
    CALL SetAPen&(rp&,Ø)
    CALL RectFill&(rp&,2Ø,2Ø,155,145)
    CALL RectFill&(rp&,165,2Ø,3ØØ,145)
    CALL SetAPen&(rp&,3)
    CALL RectFill&(rp&,159,136,161,144)
    CALL RectFill&(rp&,156,139,164,141)
    FOR i=Ø TO 3
      CALL Move&(rp&,2Ø+i,145)
      CALL Draw&(rp&,3+i,sh-1)
      CALL Move&(rp&,3ØØ-1,145)
      CALL Draw&(rp&,sb-i-3,sh-1)
```

```
      CALL Move&(rp&,20,20+i)
      CALL Draw&(rp&,3,0+i)
      CALL Move&(rp&,300,20+i)
      CALL Draw&(rp&,sb-3,0+i)
    NEXT
    CALL SetAPen&(rp&,2)
    FOR y=159 TO 181 STEP 22
      FOR x= 136 TO 290 STEP 22
        x1=RND*5:z=RND*1:IF z THEN x1=-x1
        CALL DrawEllipse&(rp&,x,y,9,9)
        CALL DrawEllipse&(rp&,x,y,1,1)
        CALL Move&(rp&,x-x1,y-5)
        CALL Draw&(rp&,x+x1,y+5)
    NEXT x,y
    Steuer:
      FOR i = 0 TO 7: READ xs(i),ys(i):NEXT
      kx1=0:ky=23 :COLOR 3
      FOR i= 45 TO -45 STEP -45
        LINE (kx1,0)-(kx1+69,69),1,bf
        cw!=COS(i*3.14/180):sw!=SIN(i*3.14/180)
        kx=kx1+35
        FOR n= 0 TO 7
          x1=xs(n)*cw!-ys(n)*sw!+kx
          y1=xs(n)*sw!+ys(n)*cw!+ky
          AREA (x1,y1)
        NEXT n
        AREAFILL :kx1=kx1+70
      NEXT
  'unteres Playfield
    FOR y= 270 TO 760 STEP 70
      x=RND*412:ins=RND*1:ins=ins*100
      CALL ClipBlit&(wrp&,ins,70,rp2&,x,y,100,70,192)
    NEXT y
    CALL ClipBlit&(wrp&,200,70,rp2&,50,830,100,70,192)
    CALL ClipBlit&(wrp&,200,70,rp2&,400,200,100,70,192)
    CALL ClipBlit&(wrp&,0,70,rp2&,200,130,100,70,192)
RETURN

Inseln:
  PRINT "Joy-Stick in Port B (2)":PRINT
  PRINT "Ihre Aufgabe ist es, die bewohnten"
```

```
  PRINT "Inseln zu versorgen. Bringen Sie dazu"
  PRINT "das Fadenkreuz ueber die mit"
  PRINT "einem >H< markierten Landeplaetze."
  PRINT "Feuerknopf = Spielabbruch"
  LINE (Ø,7Ø)-(99,139),1,bf
  CIRCLE (5Ø,1Ø5),35,3,,,.3
  PAINT (5Ø,1Ø5),3
  baum 5Ø,1Ø5
  LINE (1ØØ,7Ø)-(199,139),1,bf
  CIRCLE (15Ø,1Ø5),45,3,,,.4
  PAINT (15Ø,1Ø5),3
  baum 15Ø,1Ø5
  LINE (2ØØ,7Ø)-(299,139),1,bf
  CIRCLE (25Ø,1Ø5),49,3,2.4,.8,.5
  CIRCLE (25Ø,55),8Ø,3,4.23,5.15
  PAINT (25Ø,1Ø5),3
  CIRCLE (27Ø,1Ø5),1Ø,Ø,,,1
  PAINT (27Ø,1Ø5),Ø
  LINE (265,1Ø2)-(266,1Ø9),1,bf
  LINE (274,1Ø2)-(275,1Ø9),1,bf
  LINE (265,1Ø5)-(275,1Ø6),1,bf
  baum 23Ø,1ØØ:baum 235,115
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
### Bitte hier einfügen ###
END SUB

SUB baum (x%,y%) STATIC
  FOR i=Ø TO 2Ø
    d%=RND*7:d2%=RND*2Ø:a=RND*6:e=RND*6
    CIRCLE (x%-d2%,y%-d%),3,2,a,e
    CIRCLE (x%+d2%,y%+d%),3,2,a,e
    RANDOMIZE TIMER
    d%=RND*7:d2%=RND*2Ø:a=RND*6:e=RND*6
    CIRCLE (x%+d2%,y%-d%),3,2,a,e
    CIRCLE (x%-d2%,y%+d%),3,2,a,e
  NEXT
END SUB

DATA 25,-3,-25,-3,-25,7,-2Ø,7,-2Ø,3,+2Ø,3,2Ø,7,25,7
```

Aus Platzgründen sind einige Routinen »Speicher«, »Bitmapanlegen« und »Schirm« nur angedeutet. Entnehmen Sie diese bitte dem Vorprogramm. – Zuerst finden Sie wieder eine Aufstellung aller Variablen, die für die Ausgabe des Bildes benötigt werden. Wie Sie daraus entnehmen können, entspricht die Bitmap 1 für das PlayField A der Screen-Größe. Die Bitmap 2 dagegen ist 512 Pixel breit und 1024 Bildpunkte hoch. Beide Bitmaps enthalten 2 BitPlanes. Die folgenden Routinen bringen gegenüber dem letzten Programm nichts wesentlich Neues. Schlagen Sie bei Unklarheiten bitte dort nach. Auch in diesem Programm verzichten wir auf ein Fenster. Zur Kommunikation mit den Anwender dient der Spiele-Port. Über den Joy-Stick werden die Wünsche des Anwenders an das Programm weitergegeben.

Die Adresse der alten RasInfo-Struktur wird gerettet und dafür die Adresse der neuen RasInfo-Struktur hineingeschrieben. Mit der Intuition-Routine *RemakeDisplay* wird der Rest der Video-Ausgabe auf die geänderte ViewPort-Struktur abgestimmt. Mit *InitRastPort* wird der zweite RastPort angelegt und die Adresse der Bitmap 2 an die 4. Speicherstelle gepoked.

Die Subroutine »zeichnen« ist fast für die komplette Grafik der beiden PlayFields verantwortlich. Nach dem Festlegen der Farben mit *SetRGB4* werden die beiden Raster mit *SetRast* in den Hauptfarben gefüllt. Die beiden folgenden Rechtecke zeichnen mit der Farbe 0, also transparent, die Fenster für das Flugzeug-Cockpit. Nach diesen Fenstern zum darunterliegenden PlayField B folgt die Inneneinrichtung der Pilotenkanzel. Im einzelnen sind das ein Fadenkreuz als Zieleinrichtung, einige Linien, um einen dreidimensionalen Eindruck hervorzurufen, und 16 Rundinstrumente.

Nun kommt ein spezieller Gag des Programmes! Der Steuerknüppel des Flugzeuges bewegt sich synchron mit den Bewegungen des Joy-Sticks. Eine Bewegung nach links oder rechts dreht das Steuer ebenfalls nach links oder rechts. Eine Vorwärtsbewegung schiebt den Steuerknüppel ebenfalls nach vorne. Auch die schrägen Kombinationen werden dargestellt. Mit den Kenntnissen der Transformation von Pixel-Grafiken ist das Ganze kein besonderes Problem. Dazu wird allerdings ein Speicherbereich zum Zwischenspeichern der einzelnen Grafiken benötigt. Doch wo findet sich ein solcher Bereich? Mit den Abmessungen und der Tiefe der Bitmaps sind wir schon an die Grenze des zur Verfügung stehenden Speichers gegangen. Nun, ein Speicherbereich liegt während des Programmablaufes nutzlos und unbeachtet herum. Ich spreche vom Workbench-Screen! Aus diesem Grund finden Sie im Programm plötzlich stinknormale Grafik-Anweisungen. Diese zeichnen im Hintergrund direkt in das Basic-Programmfenster. Nach den bekannten Formeln werden die Eckpunkte des Steuerknüppels in drei verschiedenen Lagen berechnet und mit AREA und AREAFILL gezeichnet. Die unterschiedliche Auflösung des Basic-Windows wird nicht berücksichtigt, da beim späteren Kopieren die richtigen Proportionen wiederhergestellt werden.

Kommen wir nun zur Grafik für das untere PlayField. Es werden die verschiedenen Inseln in eine Super-Bitmap gezeichnet. Auch hier stellt sich die Frage nach einem Zwischenspeicher, um die Grafiken schneller zeichnen zu können. Es wird daher ebenfalls die erstmalige Zeichnung der Inseln in den Workbench-Screen verlegt. Sie finden die Erstellung der Grafik in der Subroutine »Inseln«. Dieser Programmabschnitt wurde an den Anfang des Programmes gestellt, da die PAINT-Anweisung im Hintergrund nicht funktionierte. Da am Anfang sowieso eine kleine Spielerklärung an den Anwender ausgegeben werden muß, fällt dieser Umstand nicht besonders ins Gewicht. Mit der Library-Routine *ClipBlit* werden die Inseln auf zufällige Positionen des unteren PlayFields B gebracht. Nur die beiden Inseln, die vom Anwender gesucht werden müssen, werden auf feste Positionen kopiert.

Kehren wir nun zum Hauptprogramm zurück. Dort wird in einer Schleife die Wellenform »weißes Rauschen« erzeugt, die die Voraussetzung für das Motorengeräusch ist. Die Variablen für den Programmablauf erhalten ihre Startwerte zugewiesen.

Das Programm läuft nun in der WHILE/WEND-Schleife ab. Um die einzelnen Schritte eines Schleifendurchganges besser verfolgen zu können, folgt nun die genaue Reihenfolge in Stichpunkten:

- Umwandeln der Punkte in einen String
- Ausgabe der Punkte
- restlichen Treibstoffvorrat ermitteln
- alten Balken der Treibstoffanzeige löschen
- neuen Balken der Treibstoffanzeige zeichnen
- Wenn der Joy-Stick in Y-Richtung nach vorne gedrückt wird
  Y-Position des Steuers um einen Pixel nach vorne
  Scroll-Geschwindigkeit um einen Pixel erhöhen, max. 8
- andernfalls
  Y-Position des Steuers um einen Pixel zurück
  Scroll-Geschwindigkeit um einen Pixel verringern, min. 1
- Scroll-Position Y + einen Pixel, bei Pos. 873 auf 0 zurück
- Wenn der Joy-Stick in X-Richtung bewegt wird
  Scroll-Position + oder – einen Pixel
  Wenn Joy-Stick links bewegt, dann Bild 1, andernfalls Bild 3
- andernfalls
  Bild 2 (mittlere Stellung des Steuerknüppels)
- Motorengeräusch, Tonhöhe und Lautstärke abhängig vom Tempo
- beim Überschreiten der max. PlayField-Breite auf 0 zurück
- Unterschreiten der min. PlayField-Breite in positiv wandeln

- Wenn Wartezähler abgelaufen ist
  Wenn 1. Y-Position für Kollision erreicht
    Wenn auch 1. X-Position erreicht, dann Ereignis, Warte ein
  Wenn 2. Y-Position für Kollision erreicht
    Wenn auch 2. X-Position erreicht, dann Ereignis, Warte ein
- Wenn Ereignis wahr, dann Signal, Ereignis unwahr, Punkte + 1
- Wartezähler minus eins
- neues Bild des Steuerknüppels auf neue Position kopieren
- Rx-Offset in zweite RasInfo-Struktur für PFB schreiben
- Ry-Offset in zweite RasInfo-Struktur für PFB schreiben
- Video-Display abstimmen
- ViewPort auf neue Werte scrollen
- Wenn Feuerknopf gedrückt, dann Schleifen-Variable log. unwahr
- Wenn Treibstoff verbraucht, dann Schleifen-Variable unwahr.

Diesem ausführlichen Programmablauf ist kaum noch etwas hinzuzufügen. Wie Sie sicher bereits richtig vermutet haben, soll der Wartezähler eine Mehrfacherkennung der gleichen Kollision vermeiden. Als Kollision ist in diesem Zusammenhang kein Zusammenstoß gemeint, sondern die Übereinstimmung der programmierten Werte vom Fadenkreuz des Flugzeuges mit den Landemarkierungen auf den Inseln.

Sobald die Hauptschleife des Programmes verlassen ist, wird zuerst die gerettete Adresse der RasInfo-Struktur zurückgeschrieben. Nun können die restlichen Aufräumungsarbeiten beginnen.

Eine zusätzliche Anleitung für das Programm ist nicht erforderlich. Durch die ausführliche Erklärung der Hauptschleife können Sie das Programm leicht nach Ihren eigenen Ideen abändern. In der vorliegenden Version können über 60 Punkte erreicht werden. Nachteilig ist, daß bei Beibehaltung der X-Position der Insel, ohne weiteres Zutun, bereits über 30 Punkte erreicht werden. Das könnten Sie vermeiden, indem Sie bei jedem Schleifendurchlauf auch die X-Position um einen Punkt erhöhen. Eine andere Möglichkeit ist, die beiden Inseln, während sie nicht sichtbar sind, an eine andere Position in X-Richtung zu verlegen. Mit der schnellen Kopier-Routine *ClipBlit* ist das problemlos zu bewältigen.

Ob Sie nun das Programm nach Ihren Wünschen verändern oder – noch besser – eigene Programmideen verwirklichen, ich wünsche Ihnen jedenfalls viel Spaß mit dem View-Modus *Dual PlayField.*

14.5 Ein Basic-Screen im Dual-PlayField-Modus

Inzwischen wissen Sie zur Genüge, daß der Modus *Dual PlayField* für 2 bis 6 Bit-Ebenen programmiert werden kann. Was soll uns also daran hindern, den Modus für einen normalen Basic-Screen einzusetzen? Schließlich können wir diesen bekanntermaßen bis zu einer Tiefe von 5 BitPlanes einrichten. Bei der Größe der Bitmap müssen wir uns dann allerdings auf die Bildschirmgröße beschränken. Überlegen wir einmal, was gegenüber der bisherigen Technik gleich bleibt und was geändert werden muß. Greifen wir dazu auf die Zusammenstellung zurück, die sich bereits einmal bewährt hat:

ViewPort	durch Basic (Intuition)
Screen	durch Basic (Intuition)
Window	durch Basic (Intuition)
Bitmap1	durch Basic
Bitmap2	durch Basic
RasInfo1	durch Programm
RasInfo2	durch Programm
RastPort1	durch Basic (Intuition)
RastPort2	durch Programm.

Sie sehen, daß uns die meiste Arbeit durch die normalen Basic-Anweisungen abgenommen wird. Ein Problem wird sich mit der Aufteilung in zwei Bitmaps ergeben. Bevor wir uns in Einzelheiten verlieren, gehen wir wie gewohnt der Reihe nach vor. Daß wir ohne Library-Routinen nicht auskommen, ist klar. Im einzelnen benötigen wir Routinen aus der Grafik- und der Intuition-Library.

Als nächstes überlegen wir, welche Strukturen angelegt werden müssen. Da fallen uns zuerst die beiden RasInfo-Strukturen ein. Dazu benötigen wir aber auch zwei Bitmap-Strukturen. Auch die Struktur für den zweiten RastPort wird sich nicht umgehen lassen. Den Anfang des Programmablaufes haben wir damit bereits zusammengestellt:

- Libraries öffnen
- Speicherbereiche für die Strukturen reservieren
- Screen und Window öffnen.

Nun können wir uns die benötigten Adressen des Screens, RastPorts und des ViewPorts holen. Als nächstes ändern wir den ViewPort-Modus. Sie erinnern sich sicherlich daran, daß wir das bereits beim HAM-Modus praktiziert haben. Das gleiche Unterprogramm können wir auch jetzt einsetzen. Sie sehen, es lohnt sich immer, wenn man wichtige Routinen so anlegt, daß sie für verschiedene Anwendungen eingesetzt werden können.

Nun kommt die Aufteilung der Bitmap an die Reihe. Die aktuelle Bitmap-Struktur finden wir an der 88. Speicherstelle der Screen-Struktur. Diese Struktur enthält die Anfangsadressen der einzelnen Bit-Ebenen. Je nachdem, wie die Bitmap aufgeteilt werden soll, holen wir uns die beiden Anfangsadressen. Mit diesen Adressen legen wir nun zwei neue Bitmap-Strukturen an. Das ist schon die halbe Miete. Der Rest geht wie gewohnt weiter:

- RasInfo-Struktur anlegen
- alte RasInfo-Adresse retten
- neue RasInfo-Adresse in ViewPort schreiben
- RemakeDisplay aufrufen
- RastPort für die zweite Bitmap anlegen.

Nun können schon die Grafiken in die beiden PlayFields gezeichnet werden. Dabei ist zu beachten, daß das PlayField A normale Basic-Befehle akzeptiert, das PlayField B aber nicht. In den zweiten RastPort müssen Sie daher mit den Routinen der Grafik-Library zeichnen.

Wenn Sie dann das Programm beenden, dürfen Sie natürlich auch hier nicht vergessen, die gerettete Adresse der alten RasInfo-Struktur in die richtige Speicherstelle des View-Ports zurückzuschreiben.

Probieren wir gleich an einem kleinen Programm-Beispiel aus, wie sich unser neues Wissen in der Praxis bewährt.

```
REM Durchblick  Pfad: Modus/14DUALPF/Durchblick
'P14-3
'Modus DUALPLAYFIELD, Bitmap 1=Tiefe1/Farbe Ø-1
'                     Bitmap 2=Tiefe1/Farbe 8-9
CLEAR
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):sb=32Ø
vm=1Ø24          'ViewMode  DUALPLAYFIELD
GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
SCREEN 1,sb,sh,2,1
WINDOW 2,,,Ø,1
scr&=PEEKL(WINDOW(7)+46) :rp&=WINDOW(8)
vp&=ViewPortAddress&(WINDOW(7))
CALL vpMode (vm)
bm&=PEEKL(scr&+88)
bp1&=PEEKL(bm&+8)
bp2&=PEEKL(bm&+12)
```

```
GOSUB Bitmapanlegen
GOSUB RasInfoStruktur
'alte RasInfo-Adresse retten
  rialt&=PEEKL(vp&+36)
'neue RasInfo-Adresse in ViewPort schreiben
  POKEL vp&+36,ri&
  CALL remakedisplay&
'RastPort fuer Bitmap2 anlegen
  CALL InitRastPort&(rp2&)
  POKEL rp2&+4,mb2&
GOSUB zeichnen

a=-1 :x=Ø:y=Ø :PFBA=Ø
WHILE a
  POKEW ri&+2Ø,x
  POKEW ri&+22,y
  CALL RethinkDisplay&
  CALL scrollVPort& (vp&)
  ta-=INKEY-
  IF ta-=CHR-(13) THEN a=Ø
  IF ta-=CHR-(32) THEN
    CALL vpMode(64)
    IF PFBA THEN PFBA = Ø ELSE PFBA =-1
  END IF
  IF ta-=CHR-(29) THEN y=y-1:IF y<-3Ø THEN y=-3Ø
  IF ta-=CHR-(28) THEN y=y+1:IF y>3Ø THEN y=3Ø
  IF ta-=CHR-(31) THEN x=x+1
  IF ta-=CHR-(3Ø) THEN x=x-1
WEND

'altenZustandHerstellen
  IF PFBA THEN CALL vpMode(64)
  FOR i=1 TO 1ØØØ:NEXT
  POKEW ri&+2Ø,Ø
  POKEW ri&+22,Ø
  CALL RethinkDisplay&
  CALL scrollVPort& (vp&)
  CALL vpMode(vm)
  POKEL vp&+36,rialt&
  CALL remakedisplay&
```

```
ende:
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN
    BEEP:PRINT "Speicher nicht ausreichend"
  END IF
  LIBRARY CLOSE
  WINDOW CLOSE 2
  SCREEN CLOSE 1
END
RasInfoStruktur:
  POKEL ri&,ri&+12        'Next
  POKEL ri&+4,mb1&        '1.Bitmap
  POKEW ri&+8,Ø           'RxOffset
  POKEW ri&+1Ø,Ø          'RyOffset
  POKEL ri&+12,Ø          'Next
  POKEL ri&+16,mb2&       '2.Bitmap
  POKEW ri&+2Ø,Ø          'RxOffset
  POKEW ri&+22,Ø          'RyOffset
RETURN

LibraryOeffnen:
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION ViewPortAddress&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
RETURN

Speicher:
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  mb1&=AllocRemember&(rk&,4Ø,art&) 'fuer Bitmap-Struktur1
  IF mb1&=Ø THEN fehl=2 :RETURN
  mb2&=AllocRemember&(rk&,4Ø,art&) 'fuer Bitmap-Struktur2
  IF mb2&=Ø THEN fehl=2 :RETURN
  ri&=AllocRemember&(rk&,4Ø,art&)  'fuer RasInfoStruktur
  IF ri&=Ø THEN fehl=2 :RETURN
  rp2&=AllocRemember&(rk&,8Ø,art&) 'fuer RastPort2
  IF rp2&=Ø THEN fehl=2
RETURN

Bitmapanlegen:
  bmbr1=sb                   :bmbr2=sb
  bmBytPerRow1=bmbr1/8       :bmBytPerRow2=bmbr2/8
```

```
  bmRows1=sh                    :bmRows2=sh
  POKEW mb1&,bmBytPerRow1       'Bytes/Reihe
  POKEW mb1&+2,bmRows1          'Reihen
  POKE mb1&+5,1                 'Tiefe
  POKEL mb1&+8,bp1&             'eine BitPlane
  POKEW mb2&,bmBytPerRow2       'Bytes/Reihe
  POKEW mb2&+2,bmRows2          'Reihen
  POKE mb2&+5,1                 'Tiefe
  POKEL mb2&+8,bp2&             'eine BitPlane
RETURN

zeichnen:
  x1=WINDOW(2)/2:y1=WINDOW(3)/2
  'oberes PlayField zeichnen mit Basic-Befehlen
    PALETTE 1,Ø,Ø,Ø      'Farbe schwarz
    LINE (Ø,Ø)-(sb-4,sh-4),1,bf
    FOR i= Ø TO 1Ø:CIRCLE (x1,y1-12),i,Ø,,,1.Ø5:NEXT
    COLOR Ø
    AREA (x1,y1-2Ø) :AREA (x1+1Ø,y1+2Ø)
    AREA (x1-1Ø,y1+2Ø):AREAFILL
    COLOR 1
    t1-= "Cursor-Tasten zur Steuerung"
    t2-= "RETURN=ENDE * Leertaste = wechsel"
    LOCATE 2,6:PRINT t1-
    LOCATE 3,4:PRINT t2-

  'unteres PlayField zeichnen mit Routinen der Grafik-Library
    CALL SetRGB4&(vp&,9,Ø,15,Ø)      'Farbe gruen
    FOR y = 5Ø TO 12Ø STEP 7Ø
      FOR x = 1Ø TO 25Ø STEP 6Ø
        CALL SetAPen&(rp2&,9)
        CALL RectFill&(rp2&,x,y,x+4Ø,y+6Ø)
        CALL SetAPen&(rp2&,8)
        FOR i= 5 TO 25 STEP 2Ø
          CALL RectFill&(rp2&,x+i,y+2Ø,x+1Ø+i,y+25)
        NEXT i
        CALL RectFill&(rp2&,x+18,y+23,x+22,y+35)
        CALL RectFill&(rp2&,x+5,y+37,x+36,y+55)
        CALL SetAPen&(rp2&,9)
        FOR j=37 TO 48 STEP 11
          FOR i= 7 TO 31 STEP 6
            CALL RectFill&(rp2&,x+i,y+j,x+3+i,y+j+7)
```

```
      NEXT i,j
    NEXT x,y
RETURN

SUB vpMode (mode%) STATIC
  SHARED vp&
    IF PEEKW(vp&+32) AND mode% THEN     'Modus ausschalten
       POKEW vp&+32,PEEKW(vp&+32) AND NOT mode%
    ELSE                                 'Modus einschalten
       POKEW vp&+32,PEEKW(vp&+32) OR mode%
    END IF
  CALL remakedisplay&
END SUB
```

Das Programm arbeitet mit einem Screen der Tiefe 2, der in zwei Bitmaps aufgeteilt wird. Wir haben es daher nicht mit zwei Bitmaps, sondern nur mit zwei Bit-Ebenen zu tun. Bei »Bitmapanlegen« schreiben wir die Adressen der beiden Ebenen in die Bitmap-Strukturen. Die anderen Parameter entnehmen wir den Screen-Abmessungen.

Sie sehen, der Programmablauf entspricht genau der Reihenfolge, wie wir sie vorher besprochen haben. Kommen wir daher gleich zur Subroutine »zeichnen«. In das obere PlayField A zeichnen wir mit normalen Basic-Anweisungen ein Schlüsselloch. Außerdem schreiben wir die Anwender-Informationen in das gleiche PlayField. Damit es auch etwas zu sehen gibt, zeichnen wir in das darunterliegende PlayField B zwei Reihen Monsterköpfe.

Das Hauptprogramm läuft in der WHILE/WEND-Schleife ab. Die Anweisungen des Anwenders erhält das Programm über die Tastatur. Die Cursortasten erhöhen oder verringern den Dx- und den Dy-Offset des Bildausschnittes. Diese Werte werden in die zweite RasInfo-Struktur geschrieben und der ViewPort auf die neuen Werte gescrollt. Die Betätigung der Leertaste verändert die Priorität der beiden PlayFields. Dazu wird einfach wieder unser Unterprogramm zur Änderung des View-Modus mit dem Wert 64 für PFBA aufgerufen. Mit einem einfachen Tastendruck kommt also das untere PlayField nach oben und kann genauso schnell wieder versteckt werden. Die RETURN-Taste läßt die Schleifen-Variable *a* unwahr werden. Das Programm wird beendet.

Dazu müssen wir jedoch den alten Zustand wieder herstellen. Zuerst wird das Bit für PFBA gelöscht, falls es bei Programmende gesetzt war. Dann wird der ViewPort wieder in die Null-Position für *Dx* und *Dy* gescrollt. Ganz wichtig ist, wie bereits mehrfach gesagt, das Zurückschreiben der Adresse der geretteten RasInfo-Struktur. Die Speicherfreigaben, das Schließen der Libraries, des Fensters und des Bildschirmes beenden endgültig das Programm.

Das Beispiel hat uns bewiesen, daß auch mit einem normalen Basic-Screen im Dual-PlayField-Modus programmiert werden kann. Einschränkungen gibt es bei der Anzahl und der Größe der Bitmaps. Dem steht der große Vorteil der normalen Kommunikation zwischen Programm und Anwender gegenüber.

Grafik mit AMIGA-BASIC

Spezial

4. Teil

Commodore Sachbuch

Kapitel 15

Icons

Erinnern Sie sich noch an den Tag, an dem Sie Ihren Amiga das erste Mal eingeschaltet haben? An Ihre erste Begegnung mit der grafischen Benutzerschnittstelle, der Workbench? Sicherlich waren Sie begeistert, wie leicht die einzelnen Funktionen mit der Maus aufgerufen werden konnten. Die kleinen Bilder der Icons sagten Ihnen mehr als tausend Worte. Inzwischen ist das alles für Sie so selbstverständlich geworden, daß Sie kaum noch einen Gedanken daran verschwenden.

Sicherlich haben Sie sich für Ihre Programme einige hübsche oder aussagefähige Icons mit dem Programm *Icon-Ed* erstellt. Dabei mußten Sie allerdings auf die wirklich effektvollen Doppelbild-Icons verzichten. Denn dafür ist das Werkzeug *Icon-Ed* nicht ausgelegt. Auch begrenzt dieses Programm die Größe der Icons. Für manche Zwecke will man aber ein größeres Bild zeigen. Vielleicht haben Sie sogar einen Editor, der das alles kann. Das ist natürlich gut für Sie. Aber auch dann können bei Ihnen Wünsche geweckt werden, die Ihr Editor nicht erfüllen kann. Dann ist es gut, wenn man weiß, wie die Icons programmiert werden können. Nun, in diesem Kapitel erfahren Sie, wie Sie Doppelbild-, Riesen- und andere Icons programmieren können. Ein eigener Icon-Editor, den Sie dann auch nach Ihren Wünschen verändern können, finden Sie im letzten Kapitel des Buches. An dieser Stelle eine Warnung vorweg. Jedes Doppelbild- und Riesen-Icon braucht natürlich mehr Speicherplatz als ein einfaches Icon. Auch eine längere Ladezeit beim Öffnen eines Windows müssen Sie dabei in Kauf nehmen. Besonders effektvolle Icons haben also ihren Preis.

15.1 Die Icon-Datei

Sämtliche Daten eines Icons finden sich in einer .info-Datei. Die Icon-Daten für das Programm *Versuch* sind also in der Datei *Versuch.info* untergebracht. Bei Ihren Exkursionen durch die Inhaltsverzeichnisse Ihrer Disketten sind sie Ihnen bestimmt aufgefallen. Wenn Sie ein Icon nach Ihren Wünschen ändern oder neu anlegen wollen, brauchen Sie daher nur eine neue .info-Datei erstellen. Daß es sich dabei nicht nur um die reinen Bilddaten des Piktogrammes handeln kann, steht außer Frage. Schließlich

muß irgendwo hinterlegt sein, wie groß bei einer Schublade (Directory-Icon) das Fenster sein soll, und wo es auf dem Workbench-Screen positioniert werden soll. Diese .info-Files sind also eine wichtige Angelegenheit. Sie sind der Mittelpunkt der Kommunikation zwischen Anwender und Workbench. Hier sollte auch eine kleine Anmerkung angebracht sein. Darin, daß jeder Typ von Icon eine X-beliebige Zeichnung erhalten kann, liegt natürlich auch eine Gefahr. Die Icons sollten immer so gezeichnet werden, daß der Anwender auch noch erkennen kann, um welche Art von Icon es sich handelt. So ist es zum Beispiel wenig sinnvoll, den Mülleimer so zu zeichnen, daß er für eine Schublade gehalten werden kann. Ein Disketten-Icon sollte auch als Diskette zu erkennen sein. Eine Konfusion beim Anwender sollte also auf jeden Fall vermieden werden.

Alle Informationen über die Grafik, deren Lage auf der Workbench, die Art des Icons und vieles mehr sind in Strukturen und Datenblöcken abgelegt. Bevor wir auf Einzelheiten eingehen, sehen Sie sich erst die Reihenfolge der Daten im Zusammenhang an. Diese Reihenfolge darf natürlich in dem String der .info-Datei nicht verändert werden.

15.1.1 Zusammenfassung und Reihenfolge der .info-Datei

Struktur DiskObject	immer erforderlich
Struktur DrawerData	nur bei den Icon-Typen 1,2 und 5
Struktur Image	immer erforderlich
Daten Ebene 1	immer erforderlich
Daten Ebene 2	immer erforderlich
Struktur Image Selected	nur bei Doppelbild-Icon
Daten Ebene 1	nur bei Doppelbild-Icon
Daten Ebene 2	nur bei Doppelbild-Icon
String DefaultTool	nur bei den Icon-Typen 1 und 4
String ToolTypes	nur bei dem Icon-Typ 4

Fangen wir gleich mit dem wichtigsten Bestandteil der Datensammlung, der DiskObjekt-Struktur, an. Diese Liste beginnt mit einer Codezahl, genannt Magic. Damit weiß das System, daß wirklich eine .info-Datei folgt und nicht ein Programm, welches unter dieser Bezeichnung abgelegt wurde. Es folgt die Version, zur Zeit 1. Anschließend kommt eine komplette Gadget-Struktur. Die Bilddaten sind genau wie bei einem normalen Gadget abgelegt. Sie haben jedoch teilweise eine andere Bedeutung. Die Werte für die Breite und die Höhe geben den rechteckigen Bereich an, der mit dem Mauszeiger angewählt werden kann. Dieses Rechteck kann größer oder kleiner als das Bild des Icons sein. Mit den Flags legen Sie fest, was bei der Selektierung eines Icons geschieht. Drei Möglichkeiten stehen Ihnen zur Auswahl. Entweder wird der anwählbare, rechteckige Bereich invertiert dargestellt, oder nur der Bereich des Bildes. Die dritte Variante ist für uns am interessantesten. Mit diesem Flag legen wir fest, daß das Icon aus zwei Bildern besteht.

15.1.2 Icon-Flags

4 = Das selektierbare Rechteck wird invertiert
5 = Das Icon-Bild wird invertiert
6 = Das Icon besteht aus zwei Bildern.

Die beiden Zeiger auf die Image-Strukturen werden vom System versorgt. Natürlich zeigen sie nicht auf die Position im .info-String. Auch zum Auslesen der Datenfelder sind sie nicht geeignet. Die beiden Strukturen stehen in der Regel an unterschiedlichen Stellen im Speicher. Bei der Berechnung der Größe eines .info-Strings können Sie sich nur auf die Werte für die Breite und die Höhe in der Image-Struktur verlassen. Beim Zusammenstellen eines neuen .info-Strings genügt die Eingabe irgendeines Pseudowertes (größer als Null!).

In der DiskObject-Struktur folgt nach der Gadget-Struktur die Type des Icons. Wie Sie aus der ersten Übersicht über die Reihenfolge sehen können, hängt die ganze Zusammensetzung des .info-Strings von der Type des Icons ab.

15.1.3 Objekt-Typen

1 =	DISK	Disketten-Icon
2 =	DRAW	Directory-Icon (Schubladen-Icon)
3 =	TOOL	Programm-Icon
4 =	PROJECT	Datei-Icon
5 =	GARBAGE	Mülleimer-Icon

Neben diesen fünf Typen gibt es noch zwei weitere, zum Beispiel das Kick-Icon. Für unsere Zwecke sind sie jedoch nicht interessant. Die meisten der genannten Namen der Icon-Typen sind so aussagefähig, daß keine weitere Erklärung notwendig ist. Lediglich die Unterscheidung zwischen einem Programm- und einem Datei-Icon ist nicht ganz so einfach. Ein Programm-Icon steht für ein Maschinen-Programm, z.B. die Programme *AmigaBasic* und *IconEd.* Ein Datei-Icon steht natürlich zuerst für eine Datei, aber auch Ihre Basic-Programme gehören dazu. Die Änderung des Icon-Types ist nur schwer möglich. Gleichzeitig müßten dabei eine Reihe von Daten geändert werden. Außerdem ändert sich, wie bereits erwähnt, die Zusammensetzung der ganzen Strings.

Bevor wir auf die restlichen Datenfelder der Struktur näher eingehen, finden Sie nachfolgend die vollständige Struktur im Zusammenhang.

15.1.4 Struktur DiskObject

Länge	Bezeichnung	Adresse	
Wort	Magic	0	Codezahl $e310
Wort	Version	+2	Versions-Nr. für zukünftige Änderungen
Struktur	Gadget		
Zeiger	NextGadget	+4	nächstes Gadget
Wort	LeftEdge	+8	linke Kante
Wort	TopEdge	+10	obere Kante
Wort	Width	+12	Breite
Wort	Height	+14	Höhe
Wort	Flags	+16	
Wort	Activation	+18	Ausführung $3
Wort	GadgetType	+20	Gadget-Typ $1
Zeiger	GadgetImage	+22	zeigt auf Image-Struktur von Bild 1
Zeiger	GadgetImage	+26	zeigt auf Image-Struktur von Bild 2
Zeiger	GadgetText	+30	
Langwort	MutualExcl.	+34	unberücksichtigt
Zeiger	SpecialID	+38	
Wort	GadgetID	+42	
Zeiger	UserData	+44	
Byte	Type	+48	Typ des Disk-Objektes
Zeiger	DefaultTool	+50	zeigt auf DefaultTool
Zeiger	ToolTypes	+54	zeigt auf ToolTypes
Langwort	CurrentX	+58	enthält die aktuelle X-Position
Langwort	CurrentY	+62	enthält die aktuelle Y-Position
Zeiger	DrawerData	+66	zeigt auf die Struktur DrawerData
Zeiger	ToolWindow	+70	zeigt auf ToolWindow
Langwort	StackSize	+74	Größe des Stacks

DefaultTool ist ein String, der nur bei den Typen 1 und 4 benötigt wird. Beim Aktivieren eines Icons vom Typ PROJEKT wird das dem Programm zugehörige *DefaultTool* zuerst geladen. Ist dagegen das Icon vom Typ DISK, so wird das DiskCopy-Programm aufgerufen, wenn das Icon die Quelle für eine Kopie ist. Die Zeichenkette beginnt mit einem Langwort, welches die Länge des Strings enthält. Die folgende Zeichenkette ist mit Null abgeschlossen. Das *DefaultTool* eines Basic-Programmes enthält die Zeichenkette »:AmigaBASIC«. Wird dieses Icon selektiert, so wird zuerst Amiga-Basic geladen und dann das Programm. Alle Varianten dieses Strings sind für unsere Programmierung nicht interessant. Das heißt, das Programm wird so aufgebaut, daß wir diese Informationen nicht einzeln verarbeiten müssen. Das gilt gleichermaßen für die *ToolTypes*.

Die beiden nächsten Datenfelder enthalten die Position des Icons im Fenster. Findet das System die Werte $80000000 vor, so wird eine freie Stelle für das Icon gesucht. Es folgen die beiden Zeiger auf die DrawerData-Struktur und auf den String *ToolWindow*. Das letzte Datenfeld enthält die Stacktiefe. Bei einem Wert von Null wird automatisch $1000 eingesetzt. Ältere Icons zeigen noch den Wert $1000. Wird beim Selektieren eines Icons ein Fenster geöffnet, so muß der .info-String eine DrawerData-Struktur enthalten. Diese Struktur ist eine ganz normale NewWindow-Struktur, die durch die aktuelle Window-Position ergänzt wird. Wollen Sie das normale Fenster etwas anders gestalten, so brauchen Sie nur die Struktur entsprechend zu verändern.

15.1.5 Struktur DrawerData

Länge	**Bezeichnung**	**Adresse**	
Wort	LeftEdge	+78	linke Kante des Fensters
Wort	TopEdge	+80	obere Kante des Fensters
Wort	Width	+82	Breite des Fensters
Wort	Height	+84	Höhe des Fensters
Byte	DetailPen	+86	DetailPen
Byte	BlockPen	+87	BlockPen
Langwort	IDCMPFlags	+88	IDCMP-Flags
Langwort	Flags	+92	Flags
Zeiger	FirstGadget	+96	Zeiger auf das erste Gadget
Zeiger	CheckMark	+100	Z auf Image für Markierungshaken
Zeiger	Title	+104	Zeiger auf Text für Fenster-Titel
Zeiger	Screen	+108	Zeiger auf Screen
Zeiger	Bitmap	+112	Zeiger auf Bitmap (Refresh)
Wort	MinWidth	+116	min. Breite des Fensters
Wort	MinHeight	+118	min. Höhe des Fensters
Wort	MaxWidth	+120	max. Breite des Fensters
Wort	MaxHeight	+122	max. Höhe des Fensters
Wort	Type	+124	Type $1
Langwort	CurrentX	+128	enthält die aktuelle X-Position
Langwort	CurrentY	+132	enthält die aktuelle Y-Position

Die Werte, die Sie in der Spalte *Adresse* finden, geben die Position im .info-String wieder. Damit wissen Sie das Wichtigste, was für den Aufbau eines .info-Strings benötigt wird. Beachtet werden muß dabei hauptsächlich die Reihenfolge der einzelnen Informationsblöcke, wie sie aus der ersten Tabelle dieses Kapitels *Zusammenfassung und Reihenfolge* der .info-Datei, ersichtlich ist. Zum besseren Verständnis folgt nun die komplette .info-Zeichenkette eines ganz normalen Basic-Programm-Icons.

```
                            DiskObject-Struktur
e3100001           0/2      Codezahl/Version
00000000           4        nächstes Gadget
009a0035           8/10     linke Kante/obere Kante
00290014           12/14    Breite/Höhe
0005               16       Flag 5=Das Icon-Bild wird invertiert
00030001           18/20    Activation -3/Gadget-Typ -1
0000c580           22       Zeiger auf Image-Struktur von Bild 1
00000000           26       Zeiger auf Image-Struktur von Bild 2
00000000 00000000  30/34    GadgetText/MutualExclude
00000000 0000      38/42    SpecialID/GadgetID
00000000           44       UserData
0488               48/49    Typ des Disk-Objektes 4=PROJECT/Dummy
0000adc0 00000000  50/54    DefaultTool/ToolTypes
00000088 0000002a  58/62    aktuelle X-Position/aktuelle Y-Position
00000000 00000000  66/70    DrawerData/ToolWindow
00001000           74       Größe des Stacks

                            Image-Struktur Bild 1
00000000           78/80    linke Kante/obere Kante
00290014           82/84    Breite/Höhe
0002               86       Tiefe
000282b8           88       Zeiger auf BitPlanes des Bildes
0300               90/91    PlanePick/PlaneOnOff
00000000           92       nächste Image-Struktur

                            BitPlanes
00000000 000019ff  96       Bitebene 1
ffe60000 1fffffe7           (60 Worte)
800019ff ........
........ 00000000
7ffffffe 00006630           Bitebene 2
00198000 603ffff8           (60 Worte)
60006600 ........
........ ffffff80

                            DefaultTool
0000000c                    Länge des Strings = 12
3a416d69 67614241           :AmigaBA

53494300                    SIC
```

Mit diesem Beispiel wollen wir den theoretischen Teil abschließen und uns der praktischen Seite zuwenden. Wollen Sie sich noch einige weitere Beispiele ansehen, so lesen Sie einfach einige .info-Dateien von unterschiedlichen Icons ein und schauen sich das Ergebnis auf dem Bildschirm, oder besser ausgedruckt an.

15.2 Text-Icon

Nehmen wir einmal an, Sie wollen für sich oder für einen Anwender eine Mitteilung auf einer Diskette hinterlegen. Auf der Diskette befindet sich kein Amiga-Basic. Wie ist dieses Problem zu lösen? Natürlich mit einem Icon. Damit Sie auch einigermaßen Text unterbringen können, muß es ein sehr großes Icon sein. Probieren wir also, so ein Icon zu programmieren.

Dazu überlegen wir zuerst, ob wir dazu Routinen des Betriebssystems benötigen. Um das Text-Icon optisch attraktiv gestalten zu können, sollten alle Schriftarten möglich sein. Um den Inhalt des .info-Strings dauerhaft ablegen zu können, brauchen wir einen festen Speicherplatz. Damit benötigen wir die Graphics-, Diskfont- und Intuition-Library. Um das Programm nicht unnötig zu komplizieren, holen wir uns die Parameter des Anwenders mit INPUT-Anweisungen:

```
REM TextIcon  Pfad: Spezial/15Icons/TextIcon
'P15-1
CLEAR ,25000
CLEAR ,45000&
DEFINT a-z:DIM Fontn-(50),Fonth(50),text-(5)
GOSUB Textliste:PRINT text-(0)
GOSUB LibraryOeffnen
GOSUB SpeicherReservieren1 :IF fehl THEN ende
neu:
  SchriftSetzen "topaz.font",8
  GOSUB EingabeMaske :IF fehl THEN ende
  'MusterIcon anlegen
    OPEN IconName- FOR OUTPUT AS #1
      PRINT#1,"Test";
    CLOSE #1
  SchriftSetzen VorName-,VorHoehe
  CALL Move&(WINDOW(8),2,VorHoehe)
  CALL SetWindowTitles&(WINDOW(7),SADD(text-(2)),-1)
  GOSUB eingeben
  IF ta-=CHR-(130) THEN ende
  SchriftSetzen SelName-,SelHoehe
  CALL Move&(WINDOW(8),2,SelHoehe+hoehe+3)
```

```
  CALL SetWindowTitles&(WINDOW(7),SADD(text-(3)),-1)
  GOSUB eingeben
  IF ta-=CHR-(130) THEN neu

  CALL SetWindowTitles&(WINDOW(7),SADD(text-(4)),-1)
  GOSUB BildHolen   :IF fehl THEN ende
  GOSUB speichern
  CALL SetWindowTitles&(WINDOW(7),SADD(text-(5)),-1)
  taste: ta-=INKEY-:IF ta-="" THEN taste
  IF ta-=CHR-(129) THEN neu
```

Um nicht sämtliche Daten für das Icon selbst setzen zu müssen, legen wir eine Datei mit dem Namen des gewünschten Icons an. Damit wird gleichzeitig ein .info-String erzeugt, den wir im Programm lediglich ändern müssen. Das Einlesen der Schrift erfolgt in dem Unterprogramm »SchriftSetzen«, das wir uns später ansehen wollen. Wird eine Schrift eingegeben, die nicht vorhanden ist, wird die bisherige Schrift beibehalten. Je nach der ausgewählten Schrifthöhe wird der grafische Cursor mit der Library-Routine *Move* in Position gebracht. Die Routine *SetWindowTitles* gibt wieder die nötigen Informationen an den User. Die Eingabe erfolgt in der Subroutine *eingeben*. Für das zweite Icon-Bild wird die ganze Prozedur wiederholt. Nach entsprechender Anweisung des Anwenders über die Funktionstaste F1 beginnt der Speichervorgang des .info-Strings. Das Programm kann nun beendet oder ein neuer Durchgang gestartet werden.

```
ende:
  SchriftSetzen "topaz.font",8
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF fehl THEN
    ON fehl GOSUB f1,f2
    BEEP:PRINT ft-
  END IF
  ERASE Fontn-,Fonth,text-
  LIBRARY CLOSE
  CLEAR ,25000
  CHDIR ":"
END

f1:ft-="Speicher zu klein":RETURN
f2:ft-="Fehler bei den Schriftarten":RETURN
```

Ist die Eingabe beendet, wird im Unterprogramm »Schriftsetzen« der aktuelle Zeichensatz geschlossen und der ursprüngliche Zeichensatz *topaz.font* mit der Schrifthöhe 8 wiederhergestellt. Zum Schluß des Programmes wird der belegte Speicher wieder freigegeben und der Rest aufgeräumt.

```
eingeben:
  ta-=INKEY-:IF ta-="" THEN eingeben
  IF ta-=CHR-(129) OR ta-=CHR-(13Ø) THEN RETURN ELSE PRINT ta-;
GOTO eingeben

SUB SchriftSetzen (SchriftName-,SchriftHoehe%) STATIC
  SchriftName-=SchriftName-+CHR-(Ø)
  stil%=Ø:pref%=Ø
  ta&(Ø)=SADD(SchriftName-)
  ta&(1)=SchriftHoehe%*(2^16)+stil%*2^4+pref%
  altfont&=PEEKL(WINDOW(8)+52)
  IF SchriftName-="topaz.font"+CHR-(Ø) AND SchriftHoehe%<1Ø THEN
    font&=OpenFont&(VARPTR(ta&(Ø)))
  ELSE
    font&=OpenDiskFont&(VARPTR(ta&(Ø)))
  END IF
  IF font&<>Ø THEN
    CALL CloseFont&(altfont&)
    CALL SetFont&(WINDOW(8),font&)
  END IF
END SUB
```

Nun folgen die einzelnen Subroutinen und das Unterprogramm. Zu der ersten, »eingeben«, bedarf es keiner Erläuterung. In dem Unterprogramm »SchriftSetzen« legen wir zuerst in den Feld-Variablen *ta&()* die Struktur für die Text-Attribute fest. In einem IF-Block wird anschließend geprüft, ob es sich um einen Standard-Zeichensatz oder um einen Zeichensatz der Workbench-Diskette handelt. Entsprechend wird die zuständige Library-Routine zum Öffnen des Zeichensatzes aufgerufen. Mit den Routinen *Close-Font* wird der bisherige Zeichensatz geschlossen und mit *SetFont* zum Schluß der neue Zeichensatz in die RastPort-Struktur eingetragen.

```
BildHolen:
  y11=Ø:y12=hoehe+3:Xp=Ø
  AltesObjektEinlesen:
    'Struktur DiskObject  Laenge 78 Byte
    IconName-=IconName-+".info"
    OPEN IconName- FOR INPUT AS 1
      magic=CVI(INPUT-(2,1))
      Version=CVI(INPUT-(2,1))
      nextGadget&=CVL(INPUT-(4,1))
      linkeKante=CVI(INPUT-(2,1))
      obereKante=CVI(INPUT-(2,1))
```

```
      breite1=CVI(INPUT-(2,1))
      hoehe1=CVI(INPUT-(2,1))
      flags=CVI(INPUT-(2,1))
      activation=CVI(INPUT-(2,1))
      Type=CVI(INPUT-(2,1))
      GadgetRender&=CVL(INPUT-(4,1))
      SelectRender&=CVL(INPUT-(4,1))
    CLOSE #1
  'berechnen der Speichergroesse des input-Puffers
  ben=INT((breite1+15)/16)       'Breite in Worten
  ben=ben*hoehe1*4               '2 Bit-Ebenen eines Icons
  ben1=ben*2                     'Icon mit 2 Bildern
  ben1=ben1+5ØØ
  GOSUB SpeicherReservieren2 : IF fehl THEN RETURN
  offs=Ø
  OPEN IconName- FOR INPUT AS 1
    Altlaenge=LOF(1)
    WHILE NOT EOF(1)
      POKEW inp&+offs,CVI(INPUT-(2,1))
      offs=offs+2
    WEND
  CLOSE #1
  POKEW inp&+58,32768&  'freie Position im Fenster
  POKEW inp&+62,32768&
  st=78                 'Offset bis image
  str=st+2Ø             'Offset bis 1.Bitebene
  stre=str+ben+2Ø       'Offset bis 2.Bitebene
RETURN
```

Nun wird es interessant. Zur Vorbereitung des Speicherns des neuen .info-Strings lesen wir die .info-Zeichenkette ein, die wir durch eine Pseudo-Datei erzeugt hatten. Zur späteren Weiterverarbeitung werden zuerst die ersten Daten-Felder der Disk-Object-Struktur an Variablen übergeben. Allerdings benötigen wir in diesem Programm nicht alle Variablen. Anschließend wird der komplette .info-String in den reservierten Speicherbereich ab der Adresse inp& gelegt. Dabei ändern wir gleich die aktuelle X- und Y-Position in den Speicherstellen +58 und +62 auf den Wert $8000000. Damit sucht das System für das Icon einen Platz im Window.

```
speichern:
REM Zusammensetzen der info.Datei
  REM DiskObjekt Laenge 78 Byte
    obj-=""
```

```
  FOR i =Ø TO 1Ø STEP 2
    obj-=obj-+MKI-(PEEKW(inp&+i))
  NEXT
  obj-=obj-+MKI-(breite)        'Breite
  obj-=obj-+MKI-(hoehe)         'Hoehe in Zeilen
  obj-=obj-+MKI-(6)             'Flag fuer 2-Bild-Icon
  obj-=obj-+MKI-(activation)
  obj-=obj-+MKI-(Type)
  obj-=obj-+MKL-(GadgetRender&)
  obj-=obj-+MKL-(1)             'SelectRender&
  FOR i = 3Ø TO 76 STEP 2
    obj-=obj-+MKI-(PEEKW(inp&+i))
  NEXT
dra-=""                         'kein drawer
REM Image
  ima-(Ø)="" :ima-(1)=""
  FOR i =Ø TO 1
    ima-(i)=MKI-(Ø)              'linke Kante
    ima-(i)=ima-(i)+MKI-(Ø)      'obere Kante
    ima-(i)=ima-(i)+MKI-(breite) 'Breite
    ima-(i)=ima-(i)+MKI-(hoehe)  'Hoehe
    ima-(i)=ima-(i)+MKI-(2)      'Tiefe
    ima-(i)=ima-(i)+MKL-(59664&) 'Z. auf Bitplanes Pseudowert
    ima-(i)=ima-(i)+CHR-(3)      'PlanePick
    ima-(i)=ima-(i)+CHR-(Ø)      'PlaneOnOff
    ima-(i)=ima-(i)+MKL-(Ø)     'Z.nächste ImageStr. nicht gesetzt
  NEXT i
REM Bitplanes erstes Icon
REM Feldgroesse in Worten
  IcSp=INT((breite+15)/16)       'Breite in Worten
  breite2=IcSp*16                'Aufgerundet auf Wortlaenge
  c=(breite2/16)*hoehe*2
  aa=3+((breite2/16)*(hoehe+1)*2)
  pl1-=""
  DIM Icon1(aa)
  GET (Xp,y11)-(Xp+breite-1,y11+hoehe-1),Icon1
  FOR i = 3 TO c+2
    pl1-=pl1-+MKI-(Icon1(i))
  NEXT
  ERASE Icon1
REM Bitplanes zweites Icon
```

```
    pl2-=""
    DIM Icon2(aa)
    GET (Xp,y12)-(Xp+breite-1,y12+hoehe-1),Icon2
    FOR i = 3 TO c+2
      pl2-=pl2-+MKI-(Icon2(i))
    NEXT
    ERASE Icon2
  gesamt-=obj-+dra-+ima-(Ø)+pl1-+ima-(1)+pl2-
  'DefaultTool + ToolTypes aus altem Icon uebernehmen
    Rest-="":RestLaenge=Ø
    InclEbenenAlt%=stre%+ben%
    IF default& THEN
      FOR i =InclEbenenAlt TO Altlaenge-2  STEP 2
        Rest-=Rest-+MKI-(PEEKW(inp&+i))
      NEXT
    END IF
gesamt-=gesamt-+Rest-
OPEN IconName- FOR OUTPUT AS #1
  PRINT#1,gesamt-;
CLOSE #1
kText-=IconName-+".info":KILL kText-
RETURN
```

In der Subroutine »speichern« wird die neue .info-Zeichenkette zusammengesetzt. Die erste Teilzeichenkette *obj$* erhält die Werte der DiskObject-Struktur. Die ersten 20 Byte (10 Worte) werden aus dem reservierten Speicherbereich *inp&* gelesen. Aus diesem Bereich werden wir uns noch einige Daten holen. Mit dieser Methode sparen wir uns viel Arbeit. Nun werden die neuen Werte *breite* und *hoehe* an die Zeichenkette angehängt. Wie Sie sich sicherlich erinnern, legen wir damit den selektierbaren Bereich fest. Dieser könnte andere Abmessungen haben als das Bild des Icons. Der Einfachheit halber nehmen wir aber die Icon-Abmessungen. Die beiden Variablen wurden zu Beginn des Programmes über INPUT-Anweisungen übernommen. Ansonsten ist die Zusammensetzung des Strings im Programm ausreichend kommentiert. Zum Schluß wird aus den einzelnen Teil-Zeichenketten die Gesamt-Zeichenkette zusammengesetzt. Damit kann der String des Icons gespeichert werden. Dabei tritt ein kleines Problem auf. Am Anfang von »BildHolen« haben wir die String-Variable *IconName$* mit +».info« zum Weiterverarbeiten zusammengesetzt. Wenn wir nun unter diesem Namen die neue Icon-Datei speichern, erhalten wir eine zusätzliche Info-Datei »IconName.info.info«. Diese löschen wir mit KILL und das Problem ist beseitigt.

```
Textliste:
text-(Ø)="Bitte warten"
text-(1)=textØ-+CHR-(Ø)
text-(2)="Bitte Text 1.Bild eingeben, F1 = 2.Bild, F2 = ENDE"
text-(3)="Bitte Text 2.Bild eingeben, F1 = speichern, F2 =
Neustart"
text-(4)="Bitte warten, Icon wird gespeichert"
text-(5)="F1 = weiteres Icon,  F2 = ENDE"
FOR i=1 TO 5:text-(i)=text-(i)+CHR-(Ø):NEXT
RETURN

LibraryOeffnen:
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION OpenFont&() LIBRARY
  DECLARE FUNCTION OpenDiskFont&() LIBRARY
  DECLARE FUNCTION AvailFonts&() LIBRARY
  CHDIR ":bue"
  LIBRARY "graphics.library"
  LIBRARY "diskfont.library"
  LIBRARY "intuition.library"
RETURN

SpeicherReservieren1:
  art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  memb&=AllocRemember&(rk&,8ØØ,art&) 'fuer Schriftarten
  IF memb&=Ø THEN fehl=1
RETURN
SpeicherReservieren2:
  inp&=AllocRemember&(rk&,ben1,art&)  'fuer Icon Zwischenspeicher
  IF inp&=Ø THEN fehl=1
RETURN
```

Die Reservierung der notwendigen Speicherbereiche mußte in zwei Schritten erfolgen. Am Programmanfang konnten wir bei »SpeicherReservieren1« einen ausreichenden Speicher für die Schriftarten belegen. Da die Größe des Icons erst durch die Eingaben des Anwenders bekannt wird, kann auch der Zwischenspeicher für die Icon-Struktur erst dann mit »SpeicherReservieren2« festgelegt werden.

```
EingabeMaske:
  CLS
  PRINT " TEXT ICON"
  PRINT "===========":PRINT
  INPUT "Name des Informations-Icons:    ",IconName-
  IconName-=":"+IconName-
  GOSUB vorhandeneSchriftarten :IF fehl THEN RETURN
  FOR i=Ø TO anzahl%-1
    LOCATE i+4,pn:PRINT Fontn-(i)
    LOCATE i+4,ph:PRINT Fonth(i)
  NEXT
  LOCATE 6,1:INPUT "Schriftname Icon-Vorderseite:   ",VorName-
  LOCATE 7,1:INPUT "Schrifthoehe Icon-Vorderseite:  ",VorHoehe
  LOCATE 9,1:INPUT "Schriftname selektiertes Icon:  ",SelName-
  LOCATE 1Ø,1:INPUT "Schrifthoehe selektiertes Icon: ",SelHoehe
masse:
  LOCATE 14,1:INPUT "Icon-Breite in Pixel (max.3ØØ):",breite
  LOCATE 15,1:INPUT "Icon-Hoehe in Pixel (max.8Ø):  ",hoehe
  IF breite>3ØØ OR hoehe>8Ø THEN :BEEP
**       :PRINT "Icon wird zu gross!":GOTO masse
  CLS
  CALL SetWindowTitles&(WINDOW(7),SADD(text-(1)),-1)
  LINE (breite,Ø)-(breite+6,hoehe+hoehe+6),1,bf
  LINE (Ø,hoehe)-(breite,hoehe+2),1,bf
  LINE (Ø,hoehe+hoehe+4)-(breite,hoehe+hoehe+6),1,bf
RETURN

vorhandeneSchriftarten:
  pn=5Ø :ph=7Ø
  LOCATE 1,pn:PRINT "VORHANDENE SCHRIFTARTEN:"
  LOCATE 2,pn:PRINT "========================"
  LOCATE 3,pn:PRINT "Schriftname  Schrifthoehe"
  IF zeiger THEN RETURN
  fehl&=AvailFonts&(memb&,8ØØ,3)
  IF fehl&<>Ø THEN fehl=2:RETURN
  anzahl%=PEEKW(memb&)
  memb&=memb&+2
  FOR i=Ø TO anzahl%-1
    bez&=PEEKL(memb&+2)
    Foname-=""
    holen:
```

```
    buchstabe%=PEEK(bez&)
    IF buchstabe% THEN Foname-=Foname-+CHR-(buchstabe%) **
                    :bez&=bez&+1: GOTO holen
    Fontn-(i)=Foname-
    Fonth(i)=PEEKW(memb&+6)
    memb&=memb&+1Ø
  NEXT
  zeiger = -1
RETURN
```

In der »EingabeMaske« übernimmt das Programm zuerst den Namen des Icons und dann die Schriften für die Vorderseite und die zweite Seite des Icons. Die Abmessungen des Informations-Icons haben wir auf 300 Pixel Breite und 80 Pixel Höhe begrenzt. Das ist schon ein ganz schöner Brummer. Wünschen Sie dafür andere Werte, so ändern Sie die letzte Zeile entsprechend ab. Bei einem größeren Icon müssen Sie eventuell auch den Programmspeicher, also den zweiten CLEAR-Befehl erhöhen. Da der Bildschirm für die Eingabe benötigt wird, setzen wir Fenster-Titel für Anweisungen an den Anwender ein. Der Bereich, in dem geschrieben werden kann, wird durch breite Linien abgegrenzt. Am Ende der rechten Icon-Begrenzung kann durch RETURN zur nächsten Zeile gesprungen werden. Eine programmgesteuerte Begrenzung der Texteingabe für die Textbreite und die Texthöhe ist nicht vorgesehen.

Damit der Anwender auch weiß, welche Schriftarten ihm zur Verfügung stehen, werden in der Subroutine »vorhandeneSchriftarten« mit der Diskfont-Routine *AvailFonts* alle vorhandenen Schriften in den Speicherbereich *memb&* eingelesen. In dem rechten Viertel der »EingabeMaske« werden sie dann angezeigt.

Übrigens, das Informations-Icon, welches Sie auf der dem Buch beiliegenden Diskette finden, wurde mit diesem Editor erstellt.

Kapitel 16

Speichern

Da haben Sie nun in mühevoller Arbeit eine wunderschöne Grafik auf den Bildschirm gezaubert, und wenn dann der Rechner abgeschaltet wird, ist der Zauber für immer dahin. Nicht viel weniger Frust bereitet es, wenn die grafische Darstellung mathematischer Formeln nach einigen Stunden Laufzeit endlich das gewünschte Ergebnis zeigt, dieses Ergebnis aber nur mit dem gleichen zeitlichen Aufwand zu wiederholen ist.

Der Ausweg aus diesem Dilemma ist die Ablage der Grafiken in Dateien. Damit können sie jederzeit wieder eingelesen und gegebenenfalls auch weiterbearbeitet werden. Dazu gleich eine Warnung zu Beginn. Die Speicherung von Grafiken nimmt sehr viel Speicherplatz auf Ihren Disketten in Anspruch. Wir werden noch genauer darauf eingehen. Zur Vermeidung der Fehlermeldung »DISK FULL« ist es sicher besser, sich vorher den noch verfügbaren Speicherplatz mit der Option INFO des Workbench-Menüs zu holen.

In diesem Kapitel werden Sie einige Möglichkeiten der Speicherung von Grafikbildern kennenlernen. Jede Methode hat natürlich ihre Vor- und Nachteile. Mit der meiner Meinung nach effektivsten Art und Weise der grafischen Datenspeicherung werden wir uns detailliert befassen.

16.1 Speichern mit GET

Wenn Sie einen Bildschirmbereich mit GET bestimmen, werden die einzelnen Pixel in einem numerischen Feld gespeichert. Die ersten drei Elemente des Feldes enthalten die Werte für die Breite, Höhe und Tiefe der Grafik. Anschließend folgen die Grafikdaten in Wortlänge (kurze Ganzzahl = 2 Byte). Jedes Bit entspricht dabei einem Pixel für eine Ebene der Grafik. Die Daten für die einzelnen Farbebenen liegen hintereinander. Zuerst kommt die erste Bitebene, dann die zweite etc., je nachdem, welche Tiefe das Bild hat. Ein Datenfeld mit *n* Feldelementen sieht also folgendermaßen aus:

Element 0	Datenwort enthält die Breite
Element 1	Datenwort enthält die Höhe
Element 2	Datenwort enthält die Tiefe

Element 3	1. Datenwort der ersten Bitebene
Element 4	2. Datenwort der ersten Bitebene
Element 5	3. Datenwort der ersten Bitebene
....	....
Element (n-3)/Tiefe	letztes Datenwort der ersten Bitebene
Element +1	1. Datenwort der zweiten Bitebene
Element +2	2. Datenwort der zweiten Bitebene
Element +3	3. Datenwort der zweiten Bitebene
....	....
etc.	

Dieses numerische Feld können wir aber auch in einer Datei speichern. Die einzelnen Schritte dafür könnten folgendermaßen aussehen:

File-Namen eingeben
Feld (Bild) dimensionieren
GET Bild
Datei öffnen
Schleife
Daten in Zeichenkette wandeln
Zeichenkette in die Datei schreiben
Wenn noch nicht alle Daten geschrieben dann Schleife
Datei schließen
Datenfeld freigeben.

Zum Lesen der Daten würden Sie ähnlich vorgehen. Dafür können Sie natürlich die standardmäßigen Datei-Befehle des Amiga-Basic anwenden. Bei dem folgenden Beispiel wurden allerdings Routinen der Libraries eingesetzt. Dadurch erhalten wir einen Geschwindigkeitsvorteil und können das Programm recht anwenderfreundlich aufbauen.

```
REM GETspeicher  Pfad: Spezial/16Speichern/GETspeicher
'P16-1
CLEAR ,45ØØØ& :ON ERROR GOTO fehler
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION xOpen&() LIBRARY
DECLARE FUNCTION xWrite&() LIBRARY
DECLARE FUNCTION xRead&() LIBRARY
LIBRARY ":bue/dos.library"
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"
text1-="F1=save F2=load F3=draw F4=CLS F5=end"+CHR-(Ø)
text2-="Bitte mit der Maus den Speicherbereich markieren"+CHR-(Ø)
```

```
text3-="Bitte warten"+CHR-(Ø)
DemoStart:
CALL SetWindowTitles&(WINDOW(7),SADD(text1-),-1)
taste:tas-=INKEY-:IF tas-="" THEN taste
IF tas-<CHR-(129) OR tas->CHR-(132) THEN ende
w=ASC(tas-)-128
ON w GOSUB speichernals,einlesenals,zeichnen,frei
GOTO DemoStart
zeichnen:
fa=fa+1:IF fa>3 THEN fa=1
COLOR fa:x%=RND*6ØØ:y%=RND*17Ø:RANDOMIZE TIMER
FOR i = Ø TO 1Ø
  b%=RND*6Ø:h%=RND*3Ø
  CALL DrawEllipse&(WINDOW(8),x%,y%,b%,h%)
NEXT:RETURN
frei:CLS:RETURN
```

Das Programm weist mit

```
CLEAR ,45000&
```

dem Basic-Interpreter zusätzlichen Programmspeicher zu, da die Dimensionierung für die GET-Anweisung 40 Kbyte Speicher beanspruchen kann:

```
64Ø/8 (Breite) * 256 (Höhe) * 2 (Tiefe) = 4Ø96Ø Byte
```

Die ON ERROR-Anweisung wird zur Erkennung des Fehlers Nummer 53, FILE NOT FOUND, eingesetzt. Damit wird ein Programmabbruch verhindert, wenn eine nicht existierende Datei aufgerufen wird. Es folgen die Routinen für die Libraries. Beim Label »DemoStart« werden die Funktionstasten abgefragt und entsprechend verzweigt. Die Verzweigung zum Label »zeichnen« setzt eine kleine Zeichenroutine in Aktion.

```
einlesenals:    GOSUB namelesen
einlesen:       offen&=Ø:rk&=Ø
  CALL SetWindowTitles&(WINDOW(7),SADD(text3-),-1)
  IF filename-="" THEN filename-="Namenlos"
  fina-=filename-+CHR-(Ø)
  OPEN"I",#1,filename-:laenge&=LOF(1):CLOSE#1
  offen&=xOpen&(SADD(fina-),1ØØ5)
  IF offen&=Ø THEN Fehler1
  art&=3+(216):rek&=Ø:rk&=VARPTR(rek&)
  adr&=AllocRemember&(rk&,laenge&+8,art&)
  IF adr& = Ø THEN fehler2
  lng&=laenge&/4: DIM Bild&(lng&)
```

```
  lese&=xRead&(offen&,adr&,VARPTR(laenge&))
  FOR i=Ø TO lng&-1
    Bild&(i)=PEEKL(adr&+(i*4))
  NEXT
  PUT (Ø,Ø),Bild&,PSET :ERASE Bild&
schluss:
  IF offen& THEN CALL xClose&(offen&)
  IF rk& THEN CALL FreeRemember&(rk&,-1)
RETURN
```

Nun kommen wir zum ersten von zwei Hauptteilen des Programmes, zum Einlesen der Daten eines gespeicherten Bildes. Dort wird zuerst mit LOF(1) die Länge des Files #1 ermittelt und an die Variable *laenge&* übergeben. An dieser Stelle würde die ON ERROR-Anweisung auch erkennen, wenn eine Datei nicht vorhanden ist. Mit der DOS-Routine *xOpen&* wird die Datei geöffnet. Es folgt eine Reservierung eines Speicherbereiches ab der Speicherstelle *adr&* in der Größe von *laenge&*. Zur Dimensionierung der Feldvariablen *Bild* als lange Ganzzahl (&=4 Byte) werden die Bytes aus *laenge&* durch 4 dividiert. Mit *xRead&* wird die komplette Datei in den Speicherbereich ab *adr&* gelesen. Das geht sehr schnell. Länger dauert die Zuweisung der eingelesenen Daten auf die einzelnen Feldelemente des Feldes *Bild&* in der nachfolgenden FOR/NEXT-Schleife. Mit PUT wird das Bild auf den Bildschirm gebracht.

Das war schon alles, was zum Einlesen eines Bildes benötigt wird. Abgeschlossen wird die Subroutine mit den wichtigen Befehlen zum Freigeben der Speicherbereiche. ERASE gibt die Feldvariable *Bild&*, *FreeRemember&* den Speicherbereich ab *adr&* frei und *xClose&* schließt die Datei.

```
speichernals:    GOSUB namelesen
speichern: rk&=Ø
  CALL SetWindowTitles&(WINDOW(7),SADD(text2–),-1)
  WHILE MOUSE(Ø)=Ø :WEND
  x1=MOUSE(3):y1=MOUSE(4):tiefe=2
  WHILE MOUSE(Ø)<>Ø
    muss=MOUSE(Ø):x2=MOUSE(1):y2=MOUSE(2)
    CALL SetDrMd&(WINDOW(8),3)
    LINE (x1,y1)-(x2,y2),3,b:LINE (x1,y1)-(x2,y2),3,b
    CALL SetDrMd&(WINDOW(8),1)
  WEND
  CALL SetWindowTitles&(WINDOW(7),SADD(text3–),-1)
  IF x1>x2 THEN SWAP x1,x2
  IF y1>y2 THEN SWAP y1,y2
  wbi&=6+(y2-y1+1)*2*INT((x2-x1+16)/16)*tiefe/4
```

```
  DIM Bild&(wbi&)
  GET (x1,y1)-(x2,y2),Bild&
  IF filename-="" THEN filename-="Namenlos"
  fina-=filename-+CHR-(Ø)
  OPEN"O",#1,filename-:CLOSE#1
  offen&=xOpen&(SADD(fina-),1ØØ5)
  IF offen&=Ø THEN Fehler1
  FOR i=Ø TO wbi&-1
    schreib&=xWrite&(offen&,VARPTR(Bild&(i)),4)
  NEXT i
ERASE Bild&:GOTO schluss
```

Auch das Speichern des Bildes ist nicht schwierig. Zuerst ermitteln wir durch einige Maus-Abfragen die Größe und die Lage des zu speichernden Bereiches auf dem Bildschirm. Damit der Anwender das Rechteck in jede beliebige Richtung zeichnen kann, fragen wir die Variablen *x1, y1, x2* und *y2* ab, ob sich negative Werte ergeben können. Da das zum Programmabbruch führen würde, drehen wir in einem solchen Fall mit SWAP die Werte einfach um. Die Berechnung der Speichergröße *wbi&*, zur Dimensionierung der Feldvariablen *Bild&*, erfolgt exakt mit der Formel, die Ihnen von der Programmierung der GET-Anweisung bekannt ist. Nach der Dimensionierung speichern wir mit GET das Bild in der Feldvariablen *Bild&*. Warum die OPEN-Anweisung des Basic folgt, wird im nächsten Kapitel verraten.

Nach dem Öffnen mit *xOpen&* werden in einer Schleife die einzelnen Felder der Feldvariablen *Bild&* mit *xWrite&* in die Datei geschrieben. Bei einem großen Bild dauert das Schreiben jedes einzelnen Feldelementes eine ganze Weile. Auch hier geben wir zum Schluß den Speicher wieder frei und schließen die Datei.

```
ende:END
namelesen:
  filename-=""
  tt-="Bitte Filenamen: "
  CALL SetWindowTitles&(WINDOW(7),SADD(tt-+CHR-(Ø)),-1)
  eing:ta-=INKEY-:IF ta-="" THEN eing
  IF ta-<>CHR-(13) THEN
    filename-=filename-+ta-
    zeigtitel-=tt-+filename-+CHR-(Ø)
    CALL SetWindowTitles&(WINDOW(7),SADD(zeigtitel-),-1)
    GOTO eing
  END IF
  fina-=filename-+CHR-(Ø)
  CALL SetWindowTitles&(WINDOW(7),SADD(fina-),-1)
RETURN
```

```
fehler:IF ERR<>53 THEN PRINT "Fehler "ERR:RESUME ende
Fehler1:BEEP:PRINT "Datei laesst sich nicht oeffnen":
**                                           GOTO schluss
fehler2:BEEP:PRINT "Speicher zu klein" : GOTO schluss
```

Zum Schluß noch eine kurze Programmbeschreibung. Alle Anweisungen, Eingaben und Mitteilungen an den Anwender erfolgen über die Titelleiste des Fensters. Die einzelnen Optionen des Programmes werden über die Funktionstasten 1 bis 5 angewählt. Die einzelnen Tasten haben dabei folgende Bedeutung:

`F1=save`
Zuerst werden Sie nach dem Filenamen der zu speichernden Datei gefragt. Wird kein Name eingegeben, wird der Dateiname *Namenlos* vom Programm vergeben. Anschließend werden Sie aufgefordert, mit der Maus den zu speichernden Bereich auszuwählen. Ein Mausklick markiert den Beginn des Speicherbereiches. Während die Maustaste niedergedrückt ist, wird der Bereich durch ein flimmerndes Rechteck angezeigt. Sobald die linke Maustaste losgelassen wird, beginnt der Speichervorgang. Die Wartezeit wird auch in der Titelleiste des Fensters gezeigt. Die Position und die Größe des Speicherbereiches können Sie selbst bestimmen. Gespeichert wird alles, nur die Speicherdauer steigt mit der Bildgröße.

`F2=load`
Auch hier wird der Filename über die Titelleiste eingegeben. Eine nicht vorhandene Datei wird durch eine Fehlermeldung angezeigt. Das Programm wird dabei nicht abgebrochen. Übrigens, sämtliche Fehlermeldungen werden direkt in das Fenster geschrieben.

`F3=draw`
Jeder Druck auf diese Taste zeichnet zehn verschiedene Ellipsen um einen Mittelpunkt. Alle Parameter werden durch Zufallszahlen ermittelt.

`F4=CLS`
Diese Taste löscht, wie erwartet, den Fensterinhalt.

`F5=end`
Die letzte Option dient dem ordnungsgemäßen Verlassen des Programmes.

Programmtechnisch ist die Methode »GET und PUT« am einfachsten zu verwirklichen. Dabei ist zu berücksichtigen, daß im Programm die Bildschirmdaten vernachlässigt wurden. Ein großer Nachteil ist der hohe Speicherbedarf bei der Programmierung, der schnell zu Problemen führen kann. Das größte Manko ist jedoch die fehlende Kompatibilität zu Bildern, die nach anderen Verfahren gespeichert wurden.

16.2 Speichern im IFF-ILBM-Standard

Im letzten Kapitel wurde die fehlende Übereinstimmung von gespeicherten Daten beklagt. Nun, diese Klage ist unbegründet. Beim Amiga wurde das verwirklicht, was bei anderen Computern früher vergeblich versucht wurde. Es wurde ein einheitlicher Standard geschaffen, den sich (fast) alle Softwarehäuser zu eigen gemacht haben.

Dieser Standard nennt sich INTERCHANGE FILE FORMAT oder abgekürzt IFF. Dieses austauschbare Datei-Format verdanken wir der der Firma Electronic Arts, einer amerikanischen Software-Firma. Der IFF-Standard bezieht sich nicht nur auf Grafiken. Es können auch Text-, Musik- und andere Dateien damit gespeichert werden. Mit einem Codewort am Anfang einer Datei werden die einzelnen Datentypen gekennzeichnet. Wir werden uns allerdings nur mit der Speicherung von Grafiken befassen.

Damit können Sie also Ihre eigenen Grafikbilder mit professioneller Software wie GraphiCraft, DeLuxePaint etc. weiterverarbeiten. Natürlich ist dadurch auch ein farbiger Druck Ihrer Grafiken mit Hilfe der Software möglich.

16.2.1 Aufbau der IFF-ILBM-Datei

Eine IFF-Datei setzt sich aus verschiedenen Bausteinen, genannt *Chunks*, zusammen. Jeder dieser Bausteine enthält einen Teil der Gesamtgrafik. So enthält der Chunk BMHD die Daten für den Aufbau der Bitmap. Der Chunk CMAP enthält die Farbwerte und im Chunk BODY sind die einzelnen Bildpunkte gespeichert. Die Anzahl der Chunks ist nicht vorgeschrieben, so können neue Entwicklungen bei der Software jederzeit integriert werden. Auch die Länge der Chunks ist variabel. Der Aufbau ist jedoch immer gleich. Zuerst kommt der Name des Chunks. Es folgt die Chunk-Länge in Bytes und schließlich die Daten bzw. der Inhalt des Chunks. Den schematischen Aufbau eines im IFF-Standard gespeicherten Bildes soll die folgende Tabelle verdeutlichen:

Bezeichnung	Größe	Typ	Inhalt
Grundbaustein:			
Kennwort	Langwort	String	»FORM«
FORM-Länge	Langwort	numerisch	Länge der restlichen Datei
FORM-Kennung	Langwort	String	»ILBM« (InterLeavedBitMap)
1. Chunk:			
Chunk-Name	Langwort	String	»BMHD« (BitMapHeaDer)
Chunk-Länge	Langwort	numerisch	20
Chunk-Inhalt	20 Byte	numerisch	siehe extra Tabelle
2. Chunk:			
Chunk-Name	Langwort	String	»CMAP« (Color-MAP)
Chunk-Länge	Langwort	numerisch	Länge »X« Byte
Chunk-Inhalt	X Byte	numerisch	siehe Erläuterung
3. Chunk:			
Chunk-Name	Langwort	String	»CAMG« Amiga Viewport-Modus
Chunk-Länge	Langwort	numerisch	4
Chunk-Inhalt	4 Byte	numerisch	Modus
4. Chunk:			
Chunk-Name	Langwort	String	»BODY«
Chunk-Länge	Langwort	numerisch	Länge »X« Byte
Chunk-Inhalt	X Byte	numerisch	siehe Erläuterung
Eventuell weitere Chunks, wie zum Beispiel den CCRT-Chunk, der die Graphicraft-Daten für den Farbzyklus enthält.			

Tabelle 16.1: Der Aufbau einer IFF-Datei

Ein Langwort ist, wie Sie sicherlich längst wissen, 4 Byte lang. Ist der Chunk-Inhalt eine ungerade Zahl von Bytes lang, muß ein Füll-Byte hinzugefügt werden, damit die weiteren Daten wieder von einer geraden Adresse geholt werden können. Sonst kommt es zu einer Fehlermeldung.

Damit haben wir erst die grobe Struktur der Daten für eine Grafik. Um programmgesteuert ein Bild laden oder ablegen zu können, fehlt uns noch der Aufbau der einzelnen Chunk-Inhalte.

16.2.2 Der Bitmap-Header-Chunk

Der BMHD-Chunk zeichnet hauptsächlich für die Bildschirmdaten verantwortlich. Neben den Maßen des gespeicherten Bildes in Bildschirmpunkten kann auch die Position auf dem Bildschirm festgelegt werden. Dadurch ergibt sich die Möglichkeit, auch Teilbilder zu speichern und zu laden. Auch die Screen-Daten, bestehend aus Breite, Höhe und Tiefe, sind in dem Chunk zu finden. Zwei Byte enthalten sogar das Verhältnis Breite zu Höhe eines Pixels. Bei einem Screen von 320 Pixel Breite ist das Verhältnis Breite zu Höhe = 10 : 11. Neben einigen Masken enthält auch eine Speicherstelle einen Zeiger, der aussagt, ob es sich um eine gepackte Grafik handelt. Ist der Wert <> Null, so ist das Bild nach einem besonderen Verfahren platzsparend abgelegt. Den kompletten Chunk finden Sie in der Tabelle 16.2.

Größe	**Inhalt**	
Wort	rowBytes	Bildbreite in Pixel
Wort	bitmap-rows	Bildhöhe
Wort	xPosition	X-Position auf dem Bildschirm
Wort	yPosition	Y-Position auf dem Bildschirm
Byte	nPlanes	Anzahl der BitPlanes (Tiefe)
Byte	masking	verwendete Maske
Byte	compression	gepackte Grafik
Byte	pad1	nicht belegt, für zukünft. Erweiterungen
Wort	transparentColor	für Maske
Byte	xAspect	Verhältniswert für Pixel-Breite
Byte	yAspect	Verhältniswert für Pixel-Höhe
Wort	pageWidth	Bildschirmbreite
Wort	pageHeight	Bildschirmhöhe

Tabelle 16.2: Der Aufbau eines BMHD-Chunks

16.2.3 Der Color-Map-Chunk

Der CMAP-Chunk ist wie fast alles bei den IFF-Dateien so angelegt, daß zukünftige Hard- oder Softwareänderungen ohne Speicherprobleme integriert werden können. Für jeden Farbanteil einer Farbe ist ein eigenes Byte reserviert. Der Amiga benötigt für einen Farbanteil jedoch nur 4 Bit. Damit lassen sich 16*16*16 verschiedene Farbkompositionen, also 4096 Farben realisieren. Bei 8 Bit pro Farbanteil, wie sie platzmäßig vorgesehen sind, könnte man 256*256*256 = 16.777.216 verschiedene Farben zusammensetzen. Die Länge des Chunk-Inhaltes richtet sich nach der Anzahl der Bitebenen:

Tiefe 2 = 4 Farben = 12 Byte Chunklänge
Tiefe 3 = 8 Farben = 24 Byte Chunklänge
Tiefe 4 = 16 Farben = 48 Byte Chunklänge
Tiefe 5 = 32 Farben = 96 Byte Chunklänge
Tiefe 6 = 64 Farben = 192 Byte Chunklänge

In dem Byte für einen Farbanteil sind die oberen 4 Bit mit den Farbwerten belegt. Die unteren 4 Byte sind zur Zeit nicht belegt. Wie man am einfachsten aus den drei Byte für die Farbbestandteile eine kurze Ganzzahl zaubert, so wie die Farbtabelle des Systems sie benötigt, ist in der Programmbeschreibung ausführlich erläutert.

16.2.4 Der Bitmap-Daten-Chunk

Der BODY-Chunk enthält nun endlich die einzelnen Grafikpunkte. Dabei sind die einzelnen Pixel wie folgt gespeichert. Zuerst kommt die erste Grafikzeile der ersten Bitebene, dann die erste Zeile der zweiten Bitebene, die erste Zeile der dritten Bitebene etc., je nachdem wieviel Bitmaps die Grafik hat. Nun folgt die zweite Grafikzeile und so weiter, bis das ganze Bild gespeichert ist. Um die Anzahl der Bytes pro Zeile zu erhalten, muß man diese durch 8 teilen, da ja 8 Pixel in einem Byte gespeichert werden können. Der Speicherbereich der eigentlichen Grafik errechnet sich daher wie folgt:

```
BODYgröße& = Bildbreite% / 8 * Bildhöhe% * Ebenen%
```

Damit Sie nicht von einer *Disk Full*-Meldung überrascht werden, sollten Sie vor dem Speichern einer Grafik den freien Speicherplatz der Diskette erfragen und mit dem errechneten Wert für den BODY-Chunk vergleichen. Einige Bytes für die anderen Chunks kommen noch dazu.

16.2.5 IFFuniversal, ein vielseitiges Programm

Programme, die den ganzen Bildschirm speichern, gibt es schon genug. Interessanter wäre da schon ein Programm, das nur den Fensterinhalt speichert und dieses Fenster beim Laden der Grafik wieder an die richtige Position setzt. Neben der Speicherersparnis für die Teilgrafiken finden sich etliche interessante Einsatzmöglichkeiten für so ein Werkzeug. Daß der Amiga auch Grafiken, größer als der sichtbare Bildschirmbereich, in Super-Bitmaps ablegt, ist eine feine Sache. Damit ist unter anderem ein echtes PlayField-Scrolling möglich. Setzt man solche Riesen-Bitmaps zum Beispiel in Spielen ein, müssen die Grafiken eingelesen werden können. Damit sind wir schon bei der zweiten Forderung an ein universelles Speicherprogramm. Wenn wir schon bei der Zusammenstellung von Wünschen sind, dürfen natürlich die Spezial-Modi des Amiga nicht fehlen. Bilder im HoldAndModify- und im ExtraHalfbrite-Modus sollten also auch gespeichert und wieder eingelesen werden können. Eine ganze Menge an Möglichkeiten stellen wir uns inzwischen vor. Zur besseren Übersicht listen wir die Wünsche einmal auf:

Bezeichnung/Modus	**Größe**	**Bemerkung**
Fenster LoRes	<Bildschirm	Teil-Bitmap
Fenster HiRes	<Bildschirm	Teil-Bitmap
Fenster LoRes-Lace		
Fenster HiRes-Lace		
LoRes	Bildschirm	Std.-Bitmap
HiRes	Bildschirm	Std.-Bitmap
LoRes-Lace	Bildschirm	Std.-Bitmap
HiRes-Lace	Bildschirm	Std.-Bitmap
ExtraHalfbrite	Bildschirm	Std.-Bitmap
HoldAndMidify	Bildschirm	Std.-Bitmap
Riesengrafik	<=1024*1024	Extra-Bitmap

Tabelle 16.3: Möglichkeiten von IFFuniversal

Ein weiterer Wunsch ist, daß dem Programm keine Parameter, wie Modus, Tiefe, Screengröße, Fensterposition, Fenstergröße etc., mitgegeben werden müssen. Das Programm muß diese Werte selbst ermitteln können. Damit das Laden und das Speichern einer Grafik nicht zur Tortur wird, sollten zur Beschleunigung des Programmes Routinen der Libraries eingesetzt werden. Dadurch kann das Einlesen eines Bildes auf wenige Sekunden beschränkt werden. Das Speichern dauert dann auch nicht recht viel länger. Nachdem wir nun genau wissen, was wir wollen, brauchen wir nur noch die Wünsche in die Tat umzusetzen. Packen wir's also an.

```
REM IFFuniversal  Pfad: Spezial/16Speichern/IFFuniversal
REM V4.11
'einlesen/speichern kompletter Bitmaps und View-Modi
'by Horst-Rainer Henning
'letzte Ueberarbeitung 7.7.88
'P16-2
CLEAR
REM Demo Anfang ####################################
REM Die Fenstergroesse kann beliebig gewaehlt werden
DemoStart:WINDOW CLOSE 1
WINDOW 1,"F1=save F2=load F3=end",(50,20)-(520,150),3
taste:tas-=INKEY-:IF tas-="" THEN taste
IF tas-=CHR-(130) THEN GOSUB einlesenals:GOTO DemoStart
IF tas-<CHR-(129) OR tas->CHR-(130) THEN END'SYSTEM
a=100:b%=WINDOW(2)-6:h%=WINDOW(3)-3
WHILE a :fa%=3*RND:COLOR fa%
FOR i=0 TO 2:x%(i)=b%*RND:y%(i)=h%*RND
```

```
AREA (x%(i),y%(i)):NEXT :AREAFILL Ø:a=a-1
WEND
filename-="":GOSUB speichern
GOTO DemoStart
REM Demo Ende ##########################################
'Dieses Programm kann an unvorbereitete Programme mit
'Merge angehaengt werden, wenn das Programmende am Ende
'des Listings steht. Bitte dann kein >END< eingeben.
'Bei Beendigung des Programmes wird das aktuelle Fenster
'durch Druecken der Returntaste unter dem Namen
'>Namenlos< abgespeichert.
'Andernfalls kann das Programm wie folgt aufgerufen werden:
'GOSUB einlesen = einlesen von Bildern im IFF-Standard
'GOSUB einlesenals = einlesen mit Namensgebung
'GOSUB speichern = speichern von Bildern im IFF-Standard
'GOSUB speichernals = speichern mit Namensgebung
'Der Name wird jeweils in der Fensterleiste eingegeben
'gepackte Grafiken und Farbrotation werden nicht verarbeitet
'
'wichtig!! Wenn das aufrufende Programm mit Libraries arbeitet
'muessen die Befehle aus dem Unterprogramm libraryauf
'ins aufrufende Programm uebernommen werden
'ausserdem ist die Variable libr%=-1 zu setzen
'
iffroutinen:ta-=INKEY-:IF ta-="" THEN iffroutinen
IF ta-=CHR-(13)THEN GOSUB speichern ELSE GOTO iffroutinen
LIBRARY CLOSE: END
```

Am Anfang des Programmes befindet sich ein kleines Zeichenprogramm, damit Sie die Programmfunktionen ausprobieren können. Wenn Sie das Programm zusammen mit unvorbereiteten Programmen laufen lassen wollen, so entfernen Sie das Demonstrationsprogramm. Weitere Erläuterungen dazu finden Sie am Ende des Kapitels. – Nach den Informationen wartet das Programm in einer Schleife auf die Betätigung von RETURN, damit es zum Abspeichern verzweigen kann. Wenn das Programm wie beschrieben an einem anderen Programm angehängt ist, wird der Fensterinhalt (oder Screen) unter dem Namen *Namenlos* gespeichert.

```
einlesenals:
GOSUB libraryauf
GOSUB namelesen
```

```
einlesen:
GOSUB libraryauf
CALL lesen
RETURN

SUB lesen STATIC
SHARED filename-,Text-
SHARED vp&,bm&,bmBytPerRow%,bmRows%
SHARED bitmap1&,bmDepth%
SHARED Xpos%,Ypos%,fbreite%,fhoehe%,farben%,feb%
SHARED scr&,scrb%,scrh%
SHARED scre&,rk&
fenst=Ø:superBM=Ø:screenOK=Ø:scoffen%=Ø:HAM=Ø:EHB=Ø
mem&=Ø:offen&=Ø:scre&=Ø:ft-=""
FOR i=Ø TO 5:bp&(i)=Ø:NEXT
meb&=Ø:mb&=Ø:rk&=Ø:Type%=Ø
  IF filename-="" THEN filename-="Namenlos"
  fina-=":Bild/"+filename-+CHR-(Ø)
  offen&=xOpen&(SADD(fina-),1ØØ5)
  IF offen&=Ø THEN Fehler1
  art&=3+(2^16)
  rek&=Ø:rk&=VARPTR(rek&)
  mem&=AllocRemember&(rk&,8ØØ,art&)
  IF mem& = Ø THEN Fehler2
  adr&=mem&
  lese&=xRead&(offen&,adr&,12)
  textholen (adr&)    'Grundbaustein
  IF Text-<>"FORM" THEN Fehler3
  textholen (adr&+8)    'Formkennung
  IF Text-<>"ILBM" THEN Fehler3
```

Das eigentliche Programm beginnt mit dem Programmteil zum Einlesen einer Grafik. Dabei stehen zwei Wahlmöglichkeiten zur Verfügung. Beim Label »einlesenals« wird über die Titelleiste des Fensters der Dateiname eingegeben. Dagegen muß beim Label »einlesen« die Stringvariable *filename$* bereits versorgt sein. Andernfalls wird vom Filenamen *Namenlos* ausgegangen. Bei beiden Wahlmöglichkeiten wird zur Subroutine »libraryauf« verzweigt. Dabei hat die Variable *libr%* als Zeiger die wichtige Aufgabe ein doppeltes Öffnen der Libraries zu verhindern. Arbeitet das aufrufende Programm auch mit Libraries, so müssen alle Funktions-Deklarationen und Öffnungs-Routinen der Subroutine »libraryauf« ins Hauptprogramm übernommen werden und die Variable *libr%* auf -1 gesetzt werden. Das klingt etwas komplizierter als es ist. Arbeitet das aufrufende Programm ohne Libraries, so brauchen Sie sich nicht darum zu kümmern.

Da es auch für die allgemeine Arbeit mit den Libraries interessant ist, will ich kurz darauf eingehen, warum ich diesen Weg gewählt habe. Die beiden Hauptbestandteile des Programmes bestehen aus zwei Unterprogrammen, eines zum Speichern und eines zum Einlesen von Grafiken. In einem Unterprogramm erlaubt aber Amiga-Basic keine Funktions-Deklarationen. Damit war der Wunsch nach einem völlig unabhängigen Lese- und Speicherprogramm nicht mehr gegeben. Eine andere Möglichkeit wäre gewesen, eine bereits geöffnete Bibliothek des Hauptprogrammes zu schließen, um doppelte Definitionen zu vermeiden. Es hätte also dafür gesorgt werden müssen, daß bei dem aufrufenden Hauptprogramm später benötigte Libraries wieder geöffnet werden. Aber damit wäre das Problem noch nicht beseitigt. Wenn außerdem mit DECLARE FUNCTION LIBRARY Maschinensprache-Routinen als aufrufbare Funktionen deklariert wurden, genügt das Schließen der Library alleine nicht. Es kommt bei der erneuten Deklaration zu einer Fehlermeldung mit Programmabbruch. Mit CLEAR wird die alte Deklaration zwar aufgehoben, allerdings werden dabei sämtliche Variablen mit auf Null gesetzt. Es bliebe daher nichts anderes übrig, als die benötigten Programmvariablen neu zu definieren. Diese Möglichkeit habe ich an einem anderen Programm aus diesem Buch auch ausprobiert. Wegen der Wartezeiten (neues Einlesen der .bmap-Dateien) konnte man bei wiederholtem Aufruf der Lade- und Speicher-Unterprogramme nicht vernünftig damit arbeiten. Denken Sie also bei Ihren eigenen Programmen an das CLEAR zu Programmbeginn, wenn Sie Funktionen deklarieren. Mit dem Programmstart RUN alleine werden die Deklarationen nicht aufgehoben. Ein sauberer Absturz des Rechners könnte folgen!

Das Unterprogramm »lesen« definiert zu Beginn eine Reihe von Variablen als globale Variablen, da das Unterprogramm andere Unterprogramme aufruft. Weitere Variablen erhalten ihre Ausgangswerte zugewiesen. Damit das Einlesen der Daten flott vonstatten geht, bedienen wir uns der *dos.library*. Einigen Routinen dieser Library ist ein x vorangesetzt worden, damit es nicht zu Konflikten mit den gleichlautenden Befehlen des Amiga-Basic kommt. Nach dem Öffnen der Datei mit *xOpen&* wird ein Speicherbereich mit *AllocRemember&* reserviert. Das ist notwendig, da die Library-Routine *xRead&* die Daten in den mit *AllocMem&* reservierten Speicherbereich ab Adresse *mem&* einliest. Das Unterprogramm »textholen« liest den Textstring aus dem reservierten Speicherbereich. Es wird geprüft, ob die Code-Wörter des Grundbausteines vom richtigen Format sind. Andernfalls wird zu den entsprechenden Fehlermeldungen verzweigt.

```
anzahl%=4
chunklesen:
lese&=xRead&(offen&,adr&,8)
chlg&=PEEKL(adr&+4)  'Chunklaenge
IF chlg&/2-FIX(chlg&/2)>Ø THEN chlg&=chlg&+1
```

```
textholen (adr&) :chunkname–=Text–
  IF chunkname–="BMHD" THEN bitmapdaten
  IF chunkname–="CMAP" THEN farbwert
  IF chunkname–="CAMG" THEN viewportmodus
  IF chunkname–="BODY" THEN bildwerte
  chunk%=chlg&
  rest&=xRead&(offen&,adr&,chunk%)
  IF lese&<Ø THEN Fehler5
chunklesen2:
IF anzahl% THEN chunklesen ELSE GOTO Ende
```

Das Hauptprogramm des Programmteiles zum Einlesen läuft bei der Sprungmarke »chunklesen« ab. Wenn Sie sich bei einzelnen Programmzeilen nicht ganz sicher sind, warum gerade jetzt nach einem String oder nach Zahlen gefragt wird, werfen Sie bitte einen Blick in das vorhergehende Kapitel. Das Einlesen der Daten geschieht genau in der Reihenfolge, wie es dort beschrieben ist. Für einen Chunk wird zuerst dessen Länge gelesen und wenn die Anzahl der Bytes eine ungerade Zahl ist, wird die Länge um ein Byte erhöht. (Damit immer von einer geraden Adresse gelesen wird.) Anschließend wird der Name des Chunks aus dem reservierten Speicherbereich geholt. Bei den vier Namen BMHD, CMAP, CAMG und BODY verzweigt das Programm zum Einlesen der restlichen Daten. Alle anderen Chunks werden überlesen und der Inhalt in der Variablen *rest&* vernichtet. Wenden wir uns nun den einzelnen Daten der Chunk-Inhalte zu.

```
bitmapdaten:
  anzahl%=anzahl%-1
  chunk%=chlg&
  lese&=xRead&(offen&,adr&,chunk%)
  bbreite%=PEEKW(adr&)         'Bild/Bitmap-Breite
  bhoehe%=PEEKW(adr&+2)        'Bild/Bitmap-Hoehe
  Xposi%=PEEKW(adr&+4)         'Abstand Bild vom linken Rand
  Yposi%=PEEKW(adr&+6)         'Abstand Bild vom oberen Rand
  ebenen%=PEEK(adr&+8)         'Tiefe
  Komprimierung%=PEEK(adr&+1Ø)
  IF Komprimierung% THEN Fehler4
  sbreite%=PEEKW(adr&+16)      'Screen-Breite
  shoehe%=PEEKW(adr&+18)       'Screen-Hoehe
  IF sbreite%>32Ø THEN modus%=2 ELSE modus%=1
  IF shoehe%>256 THEN modus%=modus%+2
GOTO chunklesen2
```

Kommen wir nun zu den Bitmap-Daten aus dem BMHD-Chunk. Auch hier können Sie sich wieder nach der entsprechenden Tabelle des vorangegangenen Kapitels richten. Die Variable *anzahl%* zählt mit, ob wir auch die vier Chunks, die wir zur Erstellung der Grafik benötigen, einlesen. Anschließend wandeln wir mit chunk% = chlg& die Chunk-Länge in eine kurze Ganzzahl um, da die Routine *xRead&* für die Anzahl der einzulesenden Daten einen Integer-Wert verlangt. Es wird der ganze Chunk in den reservierten Speicherbereich ab der Adresse *adr&* eingelesen. Aus diesem Speicherbereich holen wir zuerst die Bildbreite und die Bildhöhe. Richtiger gesagt, handelt es sich dabei um die Bitmap-Breite und um die Anzahl von Grafikzeilen der Bitmap. Wenn man diese Daten mit den Werten der Screen-Breite und -Höhe vergleicht, kann man entscheiden, ob es sich um ein gespeichertes Fenster, um einen Screen oder um eine Riesengrafik handelt.

Nun lesen wir nacheinander die X- und Y-Position und die Tiefe ein. Die X- und Y-Werte sind normalerweise auf Null gesetzt. Da wir aber auch Teilbilder bzw. Fenster speichern wollen, haben wir damit die Position der oberen linken Fensterkante. Das nächste Byte enthält einen Zeiger für die Komprimierung. Ist der Wert von *Komprimierung%* logisch wahr, so sind die Bilddaten kompakt gespeichert. Da unser Programm diese nicht lesen kann, bricht es sich in einem solchen Fall mit einer entsprechenden Fehlermeldung ab. Die Bytes 9 und 11 bis 15 (masking, padl, transparentColor, x- und y-Aspect) werden vom Programm nicht verarbeitet und daher überlesen. Zum Schluß des BMHD-Chunks wird entsprechend den Werten für die Screen-Breite und -Höhe der Modus für die SCREEN-Anweisung festgelegt.

```
farbwert:
  anzahl%=anzahl%-1
  chunk%=chlg&
  lese&=xRead&(offen&,adr&,chunk%)
  FOR f6%= Ø TO chunk%/3-1
    rot%=PEEK(adr&+(f6%*3))
    gruen%=PEEK(adr&+(f6%*3)+1)
    blau%=PEEK(adr&+(f6%*3)+2)
    rgb%(f6%)=rot%*16+gruen%+blau%/16
  NEXT
GOTO chunklesen2
```

Interessant ist das Festlegen der Farben aus den einzelnen RGB-Werten des CMAP-Chunks. Die Farbanteile *rot%*, *gruen%* und *blau%* werden nacheinander aus dem Speicherbereich geholt. Jeder dieser Farbbestandteile ist in einem Byte gespeichert und belegt dort die obersten 4 Bit. Der Wert ist daher um 4 Bit = 2^4 = 16 zu hoch. Für die Farbtafel brauchen wir die 3 Farbwerte zusammen in Wortlänge, wobei die oberen 4 Bit nicht belegt sind (0RGB). Nehmen wir den blauen Anteil, so teilen wir daher den

Chunk-Wert durch 16 und haben ihn damit an der richtigen Stelle des Farb-Wortes. Beim grünen Anteil wird der Chunk-Wert wieder durch 16 geteilt und das Resultat muß mit 16 multipliziert werden, da es an die zweite Stelle des Farb-Wortes rutscht. Der Wert kann also unverändert übernommen werden. Analog wird der Rotanteil berechnet: /16*16*16=*16.

```
viewportmodus:
  anzahl%=anzahl%-1
  chunk%=chlg&
  lese&=xRead&(offen&,adr&,chunk%)
  vpModus&=PEEKL(adr&)        'Viewport-Modus
GOTO chunklesen2
```

Der Chunk CAMG hat den ViewPort-Modus gespeichert. Für die speziellen Modi des Amiga können wir darauf nicht verzichten und speichern ihn deshalb zur weiteren Verarbeitung in der Variablen *vpModus&*.

```
bildwerte:
  anzahl%=anzahl%-1
  REM berechnen ob Fenster oder Bitmap
  body&=sbreite%/8*shoehe%*ebenen%
  IF chlg&<body& THEN fenst=-1         'Es ist ein Fenster
  IF chlg&-1>body& THEN superBM=-1     'Es ist Super-Bitmap
  IF vpModus& AND 2048 THEN HAM=-1     'Bild im HAM-Modus
  IF vpModus& AND 128 THEN EHB=-1      'Bild im EHB-Modus
  CALL screenwerte
  IF superBM THEN  BitmapAnlegen
  IF HAM OR EHB THEN CustomScreen
```

Damit kommen wir zum Wichtigsten, und zwar zu den Bilddaten aus dem BODY-Chunk selbst. Bevor wir die Daten einlesen können, müssen wir uns für die richtige Darstellung entscheiden. Als Vergleichswert berechnen wir die Chunk-Länge für den Screen und halten sie in der Variablen *body&* fest. Ist die tatsächliche Chunk-Länge kleiner als der Vergleichswert, so handelt es sich um ein gespeichertes Fenster. Ist sie dagegegen größer, so haben wir es mit einer Super-Bitmap zu tun. Das Ergebnis halten wir in den Variablen *fenst* und *superBM* fest. Bei gleicher Chunk-Länge haben wir natürlich die Daten für einen Screen zu erwarten. Nun müssen wir nur noch abfragen, ob es sich um einen Standard-Screen oder um einen Custom-Screen handelt. Mit einer logischen AND-Verknüpfung zwischen dem ViewPort-Modus (*vpModus&*) und den zugehörigen Werten für HAM und ExtraHalfBrite ist das kein besonderes Problem. Mit den entsprechenden Ergebnissen verzweigen wir zu den Sprungmarken »Bitmap-Anlegen« und »CustomScreen«, die wir später besprechen werden.

```
CALL videovergleich
IF scrb%=sbreite% AND scrh%=shoehe% AND bmDepth%=ebenen% THEN
  screenOK=-1
ELSE
  SCREEN 4,sbreite%,shoehe%,ebenen%,modus%
  scoffen%=-1
END IF
```

Nun werden die Screen-Daten der Grafik mit denen des aktuellen Bildschirmes verglichen. Stimmen die Werte überein, so braucht kein neuer Bildschirm geöffnet werden. Im anderen Fall wird ein neuer Screen mit den Werten des BMHD-Chunks geöffnet. Die Prüfung soll verhindern, daß Speicherkapazität des Rechners verschwendet wird.

Ein kleines Problem ergibt sich bei den Fenstermaßen. Bei der Umstellung auf die deutsche Fernsehnorm mit 256 Zeilen wurde zwar der Screen angepaßt, aber bei der Window-Struktur wurde etwas übersehen. Die maximale Höhe des Fensters wurde bei 200 Zeilen belassen. Dadurch läßt sich zwar mit der Maus das Fenster in den Bereich über Zeile 200 verschieben oder vergrößern, aber beim Programmieren bekommt man Schwierigkeiten. Legen Sie ein Fenster ganz oder teilweise in diesen Bereich, wird das Programm mit einer Fehlermeldung abgebrochen. Da ich Ihnen aber versprochen habe, daß das Programm Fenster im gesamten Bildschirmbereich verarbeiten kann, müssen wir einen Weg suchen, das Basic des Amiga zu überlisten. Das ist aber gar nicht so einfach, denn auch die Library-Routinen zeigen sich hierbei nicht von ihrer besten Seite.

Um mit *WindowLimits&* die maximale Fensterhöhe erhöhen zu können, muß es ja bereits existieren. Wenn es jedoch bereits in den kritischen Bereich gelegt ist, wird das Programm abgebrochen. Die Katze beißt sich also in den Schwanz. Ein gängiger, aber auch ein aufwendiger Weg wäre, ein Anwender-Fenster mit eigener Struktur zu erstellen. Wir wählen einen einfacheren Weg. Das Fenster bzw. der Bildausschnitt wird, wie beim kompletten Bildschirm auch, einfach in die Bitmap geschrieben. Da beim Speichern (siehe weiter unten) das Bild komplett mit dem Fensterrahmen gespeichert wird, ist auf Anhieb der Unterschied nicht zu erkennen. Liegt der Ausschnitt außerhalb des Fensters aus dem aufrufenden Programm, wird das darunterliegende Fenster, oder der Screen mit den Bilddaten überschrieben. Da dieser Umstand bei einem aufrufenden Programm kaum zutreffen wird, stört er uns auch nicht besonders.

```
bildwerte2:
  IF fenst THEN
    REM Fenster
    CALL screenwerte
    bytprozeile%=bbreite%/8
    diff%=Xposi%/8+(Yposi%*bmBytPerRow%)
    FOR h%=Ø TO bhoehe%-1
```

```
      FOR eb%=0 TO bmDepth%-1
        zeile&=ebeneadr&(eb%)+bmBytPerRow%*h%+diff%
        lese&=xRead&(offen&,zeile&,bytprozeile%)
    NEXT eb%,h%
    GOTO chunklesen2
   END IF
```

Da damit die vorbereitenden Arbeiten abgeschlossen sind, können ab dem Label »bildwerte2« endlich die Daten eingelesen werden. Zuerst kommt die Lese-Routine für die Teilbilder (bei gleichen Fenstermaßen) an die Reihe. Dazu ermitteln wir zuerst die Anzahl der Bytes pro Grafikzeile. In der Variablen *diff%* wird die Distanz zwischen der oberen linken Bildschirmkante und der Fensterposition berechnet. Wenn wir die X-Position durch 8 teilen (Byte-Länge) und das Ergebnis der Y-Position mit den Bytes pro Screen-Zeile multiplizieren, erhalten wir die richtige Distanz. Das Resultat haben Sie bereits beim Fenster des Demoprogrammes gesehen. Sie können ruhig das Fenster an verschiedene Positionen bringen, um die Richtigkeit der Berechnung zu überprüfen. Das Bild wird nun je Zeile der verschiedenen Ebenen eingelesen, bis die Fensterhöhe erreicht ist. Damit ist das Bild bereits fertig eingelesen.

```
  IF superBM OR HAM OR EHB THEN
    REM SuperBitmap
    vp&=scre&+44
    bm&=PEEKL(scre&+88)
    IF superBM THEN bitmap1&=mb&+8 ELSE bitmap1&=bm&+8
    FOR e%=0 TO bmDepth%-1
      ebeneadr&(e%)=PEEKL(bitmap1&+(4*e%))
    NEXT e%
    FOR h%=0 TO bhoehe%-1
      FOR eb%=0 TO ebenen%-1
        zeile&=ebeneadr&(eb%)+bmBytPerRow%*h%
        lese&=xRead&(offen&,zeile&,bmBytPerRow%)
      NEXT eb%
    NEXT h%
    GOTO chunklesen2
  END IF
REM Screen
    CALL screenwerte
    FOR h%=0 TO bmRows%-1
      FOR eb%=0 TO bmDepth%-1
        zeile&=ebeneadr&(eb%)+bmBytPerRow%*h%
        lese&=xRead&(offen&,zeile&,bmBytPerRow%)
    NEXT eb%,h%
GOTO chunklesen2
```

Der zweite Programmabschnitt liest die Spezial-Modi des Amiga ein. Für die Modi HAM und ExtraHalfBrite werden dabei die Daten in die Standard-Bitmap geschrieben. Bei einer Riesen-Grafik werden die Daten in die Anwender-Bitmap geschrieben. Das sichtbare Bild ist ein Ausschnitt aus der oberen linken Ecke der Bitmap. Das letzte Programmsegment des Labels »bildwerte2« ist schließlich für ein screenfüllendes Bild zuständig.

```
BitmapAnlegen:
'BitPlaneSpeicher
bmBytPerRow%=bbreite%/8 'Bytes per Grafikzeile
bmRows%=bhoehe%     'Grafik-Zeilen
bmDepth%=ebenen%     'Tiefe
volum&=bmBytPerRow%*bmRows%
FOR i = Ø TO ebenen%-1
  bp&(i)=AllocRaster&(bbreite%,bmRows%)
  IF bp&(i)=Ø THEN Fehler2
  CALL BltClear&(bp&(i),volum&,Ø)
NEXT
'BitMapStuktur
mb&=AllocRemember&(rk&,1ØØ,art&)
IF mb&=Ø THEN Fehler2
POKEW mb&,bmBytPerRow%     'Bytes/Reihe
POKEW mb&+2,bmRows%        'Reihen
POKE mb&+5,bmDepth%        'Tiefe
POKE mb&+4,Ø               'Flags
POKEW mb&+6,Ø              'Pfad
FOR n=Ø TO bmDepth%-1
  POKEL mb&+8+(n*4),bp&(n)        'n BitPlanes
NEXT
CALL InitBitMap&(mb&,bmDepth%,bbreite%,bmRows%)
Type%=64
CustomScreen:
bmBytPerRow%=bbreite%/8    'Bytes per Grafikzeile
bmRows%=bhoehe%            'Grafik-Zeilen
bmDepth%=ebenen%           'Tiefe
volum&=bmBytPerRow%*bmRows%
'SpeicherReservieren
meb&=AllocRemember&(rk&,5ØØ,art&)
IF meb&=Ø THEN Fehler2
'StrukturNeuerScreen
vModus%=vpModus&
```

```
POKEW meb&,Ø               'linke Ecke
POKEW meb&+2,Ø             'obere Ecke
POKEW meb&+4,sbreite%      'Breite
POKEW meb&+6,shoehe%       'Hoehe
POKEW meb&+8,bmDepth%      'Tiefe
POKE meb&+1Ø,Ø             'DetailPen
POKE meb&+11,1             'BlockPen
POKEW meb&+12,vModus%      'ViewModes
POKEW meb&+14,15+Type%     'Type
POKEL meb&+16,Ø            'TextAttr
POKEL meb&+2Ø,Ø            'Screen-Titel
POKEL meb&+24,Ø            'Gadget
POKEL meb&+28,mb&          'BitMap
scre&= OpenScreen&(meb&)
IF scre&=Ø THEN Fehler2
GOTO bildwerte2
```

Bei der Sprungmarke »BitmapAnlegen« werden die Voraussetzungen für die grafische Ausgabe der Sonder-Modi geschaffen. Für eine Riesen-Grafik wird zuerst die Anwender-Bitmap erstellt. Dazu wird der benötigte Speicherbereich reserviert und eine Bitmap-Struktur erstellt. Anschließend wird beim Label »CustomScreen« ein Anwender-Screen geöffnet. Zu dieser Sprungmarke verzweigen die beiden Modi HAM und ExtraHalfBrite. Diese kommen bis auf die Tiefe mit der normalen Bitmap aus. Zur Unterscheidung dient die Variable *Type%*. Bei einer Riesengrafik erhält sie den Wert 79 (Anwender-Bitmap=64 + Anwender-Screen=15). Die beiden anderen Modi benötigen nur den Wert 15 für den Anwender-Screen. Da der ganze Programmabschnitt im Listing ausreichend kommentiert ist, werden weitere Einzelheiten an dieser Stelle nicht wiederholt.

```
'Fehlerliste
Fehler1:BEEP:ft-= "Datei laesst sich nicht oeffnen": GOTO
**                                              schluss2
Fehler2:BEEP:ft-= "Speicher zu klein" : GOTO schluss2
Fehler3:BEEP:ft-= "keine IFF-Datei" : GOTO schluss2
Fehler4:BEEP:ft-= "gepackte Grafik!!!" : GOTO schluss2
Fehler5:BEEP:ft-= "Fehler beim Lesen" : GOTO schluss2

Ende:
  col%=2^ebenen%
  CALL LoadRGB4&(vp&,VARPTR(rgb%(Ø)),col%)
  schluss:
    tast1:ta-=INKEY-:IF ta-="" THEN tast1
```

```
  schluss2:
  IF scre& THEN CALL CloseScreen&(scre&)
  FOR i = 0 TO bmDepth%-1
    IF bp&(i) THEN CALL FreeRaster&(bp&(i),bbreite%,bmRows%)
  NEXT
  IF offen& THEN CALL xClose&(offen&)
  IF scoffen% THEN SCREEN CLOSE 4
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF ft-<>"" THEN BEEP:PRINT ft- :FOR i=1 TO 10000:NEXT
  CALL RethinkDisplay&
END SUB
```

Sind alle Daten eingelesen, so wird mit der Library-Routine *LoadRGB4&* die neue Farbtafel gesetzt. Mit einem Tastendruck des Anwenders wird das eingelesene Bild wieder gelöscht bzw. dessen Bildschirm geschlossen. Im einzelnen werden dafür beim Label »schluss« die reservierten Speicherbereiche wieder freigegeben und die Screens geschlossen. Die Datei wird mit *xClose&* ebenfalls geschlossen.

Sie sehen, daß das Ganze nicht besonders schwierig war. Schauen wir nun, ob das mit dem Speichern der Daten genauso leicht zu bewerkstelligen ist.

```
speichernals:
GOSUB libraryauf
GOSUB namelesen

speichern:
GOSUB libraryauf
CALL speich
RETURN

SUB speich STATIC
SHARED filename-
SHARED Text-
SHARED vp&,bm&,bmBytPerRow%,bmRows%
SHARED bitmap1&,bmDepth%,fw&
SHARED Xpos%,Ypos%,fbreite%,fhoehe%,farben%
SHARED scr&,scrb%,scrh%
fenst=0

  IF filename-="" THEN filename-="Namenlos"
  fina-=":Bild/"+filename-+CHR-(0)
  offen&=xOpen&(SADD(fina-),1006)
  IF offen&=0 THEN Fehler7
  IF offen& THEN CALL xClose&(offen&)
```

```
OPEN"O",#1,":Bild/"+filename-:CLOSE#1
ueberschreiben:
offen&=xOpen&(SADD(fina-),1ØØ5)

art&=3+(2^16)
rek&=Ø:rk&=VARPTR(rek&)
mem&=AllocRemember&(rk&,8ØØ,art&)
IF mem& = Ø THEN Fehler8
adr&=mem&
```

Bei den Labeln »speichern« und »speichernals« sind die gleichen Vorkehrungen für die Libraries getroffen, wie bereits beim Programmteil über das Lesen der Daten beschrieben. Überhaupt kann man vereinfacht sagen, daß beim Speichern der Daten der gleiche Weg beschritten wird wie beim Lesen der Daten. Anstelle der Library-Funktion zum Lesen tritt das Pendant zum Schreiben *xWrite&*.

Beim Öffnen der Datei mit *xOpen&* ist der accessMode wichtig. Damit bestimmen wir, ob eine bereits existierende Datei (MODE_OLDFILE=1005) oder eine neue Datei (MODE_NEWFILE=1006) geöffnet wird. Wollen Sie eine bereits bestehende Grafik überschreiben, so müssen Sie den Modus für OLDFILE eingeben. Damit dem Anwender eine zusätzliche Abfrage nach dem gewünschten Modus erspart bleibt, wird zuerst die Datei als NEWFILE angelegt. Damit soll bei einer vorhandenen Datei mit diesem Namen ein Fehler provoziert werden. Leider funktioniert die Fehlererkennung nicht immer. Außerdem kann eine visuelle Fehlermeldung die einzulesende Grafik zerstören. Eine Fehlermeldung wird vom Programm mit der Library-Routine *IoErr&* daher abgefangen (und eine bestehende Datei überschrieben). Sie müssen also selbst darauf achten, daß Sie nicht ungewollt eine Grafik überschreiben, oder zusätzlich eine andere Routine zum Abfangen des Fehlers einbauen (z.B. akustisch).

Anschließend wird die geöffnete Datei wieder geschlossen, da bei den folgenden Zeilen die Datei mit dem Modus OLDFILE wieder geöffnet werden muß (nun existiert ja die Datei). Jetzt folgt eine Programmzeile, die scheinbar so überflüssig wie ein Kropf ist:

```
OPEN"O",#1,":Bild/"+filename-:CLOSE#1
```

Sicherlich werden Sie jetzt denken: »Jetzt öffnet er die gleiche Datei nochmal, um sie sofort wieder zu schließen, und das mit Basic-Befehlen, obwohl doch alles mit DOS-Routinen programmiert wird.« Nun, gerade im DOS liegt der Hase begraben. Das DOS erstellt nämlich kein Datei-Icon. Mit der unsinnig erscheinenden Programmzeile wurde jedoch der Grafik ein Icon zugewiesen. Sie müssen also nicht erst mit dem CLI die Diskette nach möglichen Dateien durchsuchen.

```
'Pruefen ob Fenster geoeffnet
CALL screenwerte
IF fw& THEN
  CALL videovergleich
  bodygr&=fbreite%/8*fhoehe%*bmDepth%
  fenst=-1
ELSE
  bodygr&=bmBytPerRow%*bmRows%*bmDepth%
  fbreite%=bmBytPerRow%*8
  fhoehe%=bmRows%
  Xpos%=0:Ypos%=0
END IF
bytes&= 4+28+ 8+farben%*3 +8+bodygr&+12
t1-="FORM":t2-="ILBM":t3-="BMHD":t4-="CMAP"
t5-="BODY":t6-="CAMG"

'form
schreib&=xWrite&(offen&,SADD(t1-),4)
schreib&=xWrite&(offen&,VARPTR(bytes&),4)
schreib&=xWrite&(offen&,SADD(t2-),4)
IF schreib&=0 THEN Fehler6

'bmhd
a1&=20:a2&=0
tiefe%=bmDepth%*2^8    'um ein Byte versetzt
schreib&=xWrite&(offen&,SADD(t3-),4)
schreib&=xWrite&(offen&,VARPTR(a1&),4)      'chunklaenge bmhd
schreib&=xWrite&(offen&,VARPTR(fbreite%),2)
schreib&=xWrite&(offen&,VARPTR(fhoehe%),2)  'Fensterhoehe
schreib&=xWrite&(offen&,VARPTR(Xpos%),2)    'X-Position
schreib&=xWrite&(offen&,VARPTR(Ypos%),2)    'Y-Position
schreib&=xWrite&(offen&,VARPTR(tiefe%),2)   'Tiefe 1 Byte
**                                             verschoben
schreib&=xWrite&(offen&,VARPTR(a2&),4)
IF scrb%=640 THEN
  verhaeltnis%=5*2^8+11
ELSE
  verhaeltnis%=10*2^8+11
END IF
schreib&=xWrite&(offen&,VARPTR(verhaeltnis%),2)
schreib&=xWrite&(offen&,VARPTR(scrb%),2)    'Screen-Breite
schreib&=xWrite&(offen&,VARPTR(scrh%),2)    'Screen-Hoehe
```

Bis zur Bemerkung »'bmhd« wird Ihnen alles geläufig sein. Zur Speicherung der Variablen *tiefe%* wenden wir einen kleinen Trick an. Da ein einzelnes Byte mit der Routine *xWrite&* nicht so einfach zu speichern ist, machen wir eine kurze Ganzzahl daraus, indem wir den Wert um 2^8 erhöhen. Der Byte-Inhalt wird also um 8 Bit nach links geschoben und ist bei der Wort-Speicherung an der richtigen Speicherstelle.

```
  'cmap
  CALL screenwerte
  fargr&=farben%*3
  cm&=PEEKL(vp&+4)
  schreib&=xWrite&(offen&,SADD(t4-),4)
  schreib&=xWrite&(offen&,VARPTR(fargr&),4)

  FOR f%=Ø TO farben%-1
    fab%=GetRGB4&(cm&,f%)
    POKE mem&+3*f%,(fab% AND &HFØØ)/16 'isolieren u. 1 Nibbel
**                                       nach rechts
    POKE mem&+1+3*f%,(fab% AND &HFØ)   'isolieren
    POKE mem&+2+3*f%,(fab% AND &HF)*16 'isolieren u. 1 Nibbel
**                                        nach links
  NEXT
  schreib&=xWrite&(offen&,mem&,fargr&)

  'camg
  vpModus&=PEEKW(vp&+32)        'ViewPort-Modus
  cs&=4
  schreib&=xWrite&(offen&,SADD(t6-),4)
  schreib&=xWrite&(offen&,VARPTR(cs&),4)
  schreib&=xWrite&(offen&,VARPTR(vpModus&),4)
```

Die Speicherung der Farbe ist wieder etwas komplizierter. Es muß die in einem Wort gespeicherte Farbe der Farbtafel in die einzelnen Farbbestandteile zerlegt und anschließend diese in einem Byte gespeichert werden. Dabei müssen die Anteile in die oberen Nibbel der Bytes. Dazu isolieren wir zuerst den Rotanteil durch eine logische AND-Verknüpfung mit dem Wert $F00. Zur weiteren Isolierung wird der Grünanteil mit $F0 und der blaue Teil mit $F verknüpft. Je nach der Position innerhalb der kurzen Ganzzahl der Farbtafel werden die einzelnen Farbbestandteile an die Position gebracht, an der sie im Farb-Byte stehen müssen (/2^4,bleibt,*2^4). Es ist also wieder der umgekehrte Weg wie beim Einlesen der Farben.

```
  'body
  schreib&=xWrite&(offen&,SADD(t5-),4)
  schreib&=xWrite&(offen&,VARPTR(bodygr&),4)
```

```
  IF fenst THEN
    REM Fenster
    bytprozeile%=fbreite%/8
    diff%=Xpos%/8+(Ypos%*bmBytPerRow%)
    FOR h%=Ø TO fhoehe%-1
    FOR eb%=Ø TO bmDepth%-1
      zeile&=ebeneadr&(eb%)+bmBytPerRow%*h%+diff%
      schreib&=xWrite&(offen&,zeile&,bytprozeile%)
    NEXT eb%,h%
  ELSE
    REM BitMap
    FOR h%=Ø TO bmRows%-1
      FOR eb%=Ø TO bmDepth%-1
        zeile&=ebeneadr&(eb%)+bmBytPerRow%*h%
        schreib&=xWrite&(offen&,zeile&,bmBytPerRow%)
    NEXT eb%,h%
  END IF

  GOTO beenden

Fehler8:BEEP:PRINT "Speicher zu klein" : GOTO beenden
Fehler6:BEEP:PRINT "Fehler beim Schreiben" : GOTO beenden
Fehler7:BEEP:fehl%=IoErr&
  IF fehl%=2Ø2 THEN :BEEP:GOTO ueberschreiben

beenden:
  IF offen& THEN CALL xClose&(offen&)
  IF rk& THEN CALL FreeRemember&(rk&,-1)
END SUB

namelesen:
filename-=""
tt-="Bitte Filenamen: "
ti&=SADD(tt-+CHR-(Ø))
CALL SetWindowTitles&(WINDOW(7),ti&,-1)
eing:
ta-=INKEY-:IF ta-="" THEN eing
IF ta-<>CHR-(13) THEN
  filename-=filename-+ta-
  zeigtitel-=tt-+filename-+CHR-(Ø)
  CALL SetWindowTitles&(WINDOW(7),SADD(zeigtitel-),-1)
  GOTO eing
END IF
```

```
fina-=filename-+CHR-(Ø)
CALL SetWindowTitles&(WINDOW(7),SADD(fina-),-1)
RETURN

dimen:
IF dimension THEN RETURN
DIM SHARED rgb%(63),ebeneadr&(5),bp&(5)
dimension=-1:RETURN

SUB textholen (speicher&) STATIC
SHARED Text-
Text-="":FOR i = Ø TO 3
Text-=Text-+CHR-(PEEK(speicher&+i)):NEXT
END SUB

libraryauf:
GOSUB dimen
IF libr% THEN RETURN
'libr%=Ø keine Library geoeffnet
'libr%=-1 Library ist geoeffnet
libr%=-1
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION AllocRaster&() LIBRARY
DECLARE FUNCTION OpenScreen&() LIBRARY
DECLARE FUNCTION xOpen&() LIBRARY
DECLARE FUNCTION xWrite&() LIBRARY
DECLARE FUNCTION xRead&() LIBRARY
DECLARE FUNCTION GetRGB4&() LIBRARY
DECLARE FUNCTION IoErr&() LIBRARY
DECLARE FUNCTION ViewAddress&() LIBRARY
LIBRARY ":bue/dos.library"
LIBRARY ":bue/exec.library"
LIBRARY ":bue/graphics.library"
LIBRARY ":bue/intuition.library"
RETURN
```

Der Rest der zu speichernden Daten enthält keine erwähnenswerten Besonderheiten, die noch nicht besprochen wurden. Bei den Subroutinen schauen wir uns noch kurz das Eingeben des Namens über die Titelleiste bei »namelesen« an. Sämtliche Texte, die mit der Routine *SetWindowTitles&* eingegeben werden, müssen mit CHR$(0) abgeschlossen sein, damit das Ende des Textes erkannt werden kann. Mit dem ersten Text »Bitte Filenamen:« wird der Anwender zur Eingabe des Datei-Namens aufgefordert. Es wird die Tastatur abgefragt, der String der gedrückten Taste dem ersten Text hinzugefügt und

über die Titelleiste ausgegeben. Eine Korrektur der Eingabe ist nicht programmiert. Wird die RETURN-Taste gedrückt, ist die Eingabe beendet. Der eingegebene Filename erscheint nun als Fenstertext.

```
SUB videovergleich STATIC
SHARED Xpos%,Ypos%,fbreite%,fhoehe%
  Xpos%=PEEKW(WINDOW(7)+4)
  Ypos%=PEEKW(WINDOW(7)+6)
  fbreite%=PEEKW(WINDOW(7)+8)
  fhoehe%=PEEKW(WINDOW(7)+1Ø)
END SUB

SUB screenwerte STATIC
REM BitMapwerte
SHARED vp&,bm&,bmBytPerRow%,bmRows%
SHARED bitmap1&,bmDepth%,fw&,farben%
SHARED scr&,scrb%,scrh%
  v&=ViewAddress&          'View-Adresse
  vp&=PEEKL(v&)            'ViewPort-Adresse
  ras&=PEEKL(vp&+36)       'RasInfo-Adresse
  bm&=PEEKL(ras&+4)        'Bitmap-Adresse
  bmBytPerRow%=PEEKW(bm&)  'Bytes per Grafikzeile
  bmRows%=PEEKW(bm&+2)     'Grafik-Zeilen
  bmDepth%=PEEK(bm&+5)     'Tiefe
  farben%=2^bmDepth%       'Anzahl der Farben
  bitmap1&=bm&+8
  scr&=vp&-44              'Adresse Screen-Struktur
    fw&=WINDOW(7)
  IF fw& THEN
    IF PEEKL(fw&+46)=scr& THEN fw&=-1 ELSE fw&=Ø
  END IF
  scrb%=PEEKW(scr&+12)     'Breite aktueller Screen
  scrh%=PEEKW(scr&+14)     'Hoehe aktueller Screen
  FOR eb%=Ø TO bmDepth%-1
    ebeneadr&(eb%)=PEEKL(bitmap1&+(4*eb%))
  NEXT
END SUB
```

Die beiden Subroutinen »videovergleich« und »screenwerte« holen aus der Window- und der Screen-Struktur die benötigten Werte. Interessant ist davon die zweite Subroutine. Hier liegt das Geheimnis begraben, warum das Programm *IFFuniversal* auf eine Parameterübergabe der Bildwerte des aufrufenden Programmes verzichten kann. Viel-

leicht haben Sie schon einmal ein Basic-Programm zum Laden und Speichern von IFF-ILBM-Bildern gesehen, bei dem die Bildwerte über die Funktion WINDOW(7) ermittelt wurden. Nun, das funktioniert bei unserem Programm nicht. Wir laden oder speichern mit dem Programm auch Custom-Screens ohne Windows, und da liefert die Funktion WINDOW(7) rein gar nichts. Mit gar nichts meine ich den Wert Null. Damit könnte man noch zurecht kommen. Zu allem Unglück kann uns die Funktion WINDOW(7) ein positives Ergebnis liefern, obwohl kein aufrufendes Fenster vorhanden ist. Das ist zum Beispiel der Fall, wenn das Window 1 des Workbench-Screens geöffnet ist. Wir müssen also auch einen Weg finden, der uns sicher beweist, ob ein Fenster oder der gesamte Screen gespeichert werden soll. Sie erinnern sich? Diese Kernfrage haben wir auch in unserem Programm gestellt. Am Anfang des Unterprogrammes »speich« fragten wir mit

```
'Pruefen ob Fenster geoeffnet
CALL screenwerte
IF fw& THEN
```

ob die Variable *fw&* wahr ist, also das aufrufende Programm einen Fensterinhalt speichern will.

Gehen wir die Sache systematisch an. Da uns keine Basic-Funktion weiterhilft, kann uns nur eine Betriebssystem-Routine retten. Die einzig geeignete Library-Funktion ist

```
v&=ViewAddress&          'View-Adresse
```

da ihr keine Parameter mitgegeben werden müssen. Um von dieser Funktion zur Bitmap-Struktur und Screen-Struktur zu gelangen, müssen wir uns allerdings durch einige andere Strukturen vortasten. Über die View-Adresse gelangen wir an die ViewPort-Adresse. Dort finden wir die RasInfo-Adresse, welche schließlich die Bitmap-Adresse enthält. In der Bitmap-Struktur stehen ab der 8. Speicherstelle die Adressen der einzelnen BitPlanes. In der FOR/NEXT-Schleife werden sie gelesen und in der Feldvariablen *ebeneadr&()* festgehalten. Nun aber zurück zur zentralen Frage: Ist es ein Fenster, das gespeichert werden soll? Dazu ermitteln wir zuerst die Adresse der Screen-Struktur (*scr&*) über die ViewPort-Adresse. Dann fragen wir mit WINDOW(7), ob irgendein Fenster geöffnet ist und übergeben den Wert der Variablen *fw&*. Ist der Wert logisch wahr, also ein Fenster vorhanden, fragen wir, ob das Fenster zu unserem Screen gehört:

```
IF fw& THEN
  IF PEEKL(fw&+46)=scr& THEN fw&=-1 ELSE fw&=Ø
END IF
```

Die Adresse des Screens für das Fenster finden wir in der 46. Speicherstelle der Window-Struktur. Ist dort die gleiche Adresse hinterlegt wie in *scr&*, so gehört es zum gleichen Screen. Den genauen Weg können Sie anhand der Strukturen mitverfolgen, die Sie im Anhang dieses Buches finden.

Sie sehen, daß auch die Grafik-IFF-Dateien problemlos vom Basic aus programmiert werden können. Auch in der Geschwindigkeit des Ladens und des Abspeichern läßt das Basic-Programm keine Wünsche offen.

16.2.6 Arbeiten mit IFFuniversal

Mit dem Demonstrations-Programm, das dem eigentlichen Programm vorangestellt ist, können Sie die Fähigkeiten dieses Werkzeuges eindrucksvoll testen. Sie starten einfach das Programm und erhalten nach wenigen Augenblicken als Eingabefenster das verkleinerte Window 1. In der Titelleiste sind die Programmfunktionen erklärt.

```
F1=save F2=load F3=end
```

Bevor Sie ein Demo-Bild speichern, können Sie das Fenster beliebig vergrößern, verkleinern oder verschieben. Sobald die Funktionstaste 1 gedrückt ist, werden blitzschnell 100 Dreiecke, deren Eckpunkte durch Zufallszahlen ermittelt wurden, gezeichnet. Anschließend wird der Fensterinhalt unter dem Namen *Namenlos* gespeichert. Betätigen Sie anschließend F2, so werden Sie nach dem Namen der einzulesenden Datei gefragt. Die Eingabe kann über die Titelleiste des Fensters verfolgt werden. Drücken Sie RETURN, ohne Namenseingabe, so wird das Demo-Bild *Namenlos* geladen. Sie werden sehen, es erscheint genau an der gleichen Position, unter der Sie es gespeichert haben. Natürlich können Sie mit dem Demonstrations-Programm auch alle anderen Bilder einlesen. Da sämtliche Grafiken der Diskette zu diesem Buch in der Schublade *Bild* abgelegt sind, wurde in den *öffnungs*-Befehlen dieser Pfad mit aufgeführt. Sie brauchen also zum Lesen und Speichern nur den Filenamen der Grafik eingeben. Unter anderem finden Sie in dieser Schublade Grafiken im Interlace-, HAM- und ExtraHalfBrite-Modus und auch eine Grafik mit einer Super-Bitmap. Beachten Sie beim Ablegen weiterer Bilder die Kapazität der Diskette.

Wollen Sie Grafiken unter einem anderen Pfadnamen ablegen oder einlesen, so ändern Sie bitte die drei folgenden Zeilen des Programmes wie folgt:

- zu Beginn des Unterprogrammes *lesen*

  ```
  fina—=":Bild/"+filename—+CHR—(Ø)
  ```
  wird
  ```
  fina—=filename—+CHR—(Ø)
  ```

- zu Beginn des Unterprogrammes *speich*

  ```
  fina—=":Bild/"+filename—+CHR—(Ø)
  ```
  wird
  ```
  fina—=filename—+CHR—(Ø)
  ```
  ```
  OPEN"O",#1,":Bild/"+filename—:CLOSE#1
  ```
  wird
  ```
  OPEN"O",#1,filename—:CLOSE#1
  ```

Nach dieser Änderung können Sie jeden beliebigen Pfad-Namen eingeben. Damit Sie sehen, wie einfach das Speichern von Grafiken der verschiedenen View-Modi ist, finden Sie nun eine kleine Zusammenstellung, aus der Sie sehen können, wie die Demo-Bilder entstanden sind. Fangen wir mit dem einfachsten Beispiel, dem Bild *lace2__320* an. Das Programm zum Zeichnen der Grafik im Modus LACE ist recht kurz:

```
REM lace  Pfad: Spezial/16Speichern/lace
'P16-3
  SCREEN 1,32Ø,512,2,3: WINDOW 2,,,Ø,1
  FOR f=Ø TO 31Ø STEP 2
    LINE (f,2*f)-(31Ø-f,256),3 :LINE -(1Ø+f,5ØØ-f),2
  NEXT
  FOR f=Ø TO 31Ø STEP 2
    LINE (3*f,f)-(31Ø-f,256),1
    LINE -(15Ø+f/2,5ØØ-f),3 :LINE -(15+f/4,5ØØ-f/8),2
  NEXT
  filename-="lace_2_32Ø":GOSUB speichern
  WINDOW CLOSE 2: SCREEN CLOSE 1
```

An das Programm angehängt ist das Programm *IFFuniversal.* Wichtig zum Speichern ist die vorletzte Zeile. Die Grafik ist fertiggezeichnet. Gespeichert wird, bevor der Bildschirm geschlossen wird. Der Name wird an die String-Variable *filename$* übergeben. Nun wird zum Label »speichern« (im Programm *IFFuniversal*) verzweigt und das Bild wird auf Diskette gespeichert.

Bild HAMtest

Es handelt sich um die Grafik aus dem Programm *colorful.* Zuerst laden Sie das Programm und hängen mit MERGE, im Direkt-Modus, das Programm *IFFuniversal* an. Nun kopieren Sie aus der Subroutine »libraryauf« (vom Programm *IFFuniversal*) die Funktions-Deklarationen und die Öffnungs-Zeilen für die Libraries in die Subroutine »LibraryOeffnen«. Einige Zeilen sind nun doppelt vorhanden und werden gelöscht. Anschließend fügen Sie vor der WHILE/WEND-Schleife, zum Abfragen des Message-Ports, folgende Zeile ein:

```
libr%=-1:filename-="HAMtest":GOSUB speichern
```

Ist die Variable *libr%* logisch wahr, so wird im Programm *IFFuniversal* keine Library geöffnet und keine Library-Funktion deklariert. Nun brauchen Sie nur noch das Programm starten und die Grafik wird gespeichert.

Bild EHBtest

Die Grafik zeigt das Bild aus dem Programm *shadow.* Außer einem anderen Namen für *filename$* können Sie exakt wie im letzten Beispiel vorgehen.

Bild SuperBMtest
Diese Grafik zeigt die Kirche aus dem Programm *BigPlane* auf 1024 * 1024 Grafikpunkten. Die Programmzeile zum Speichern fügen Sie dieses Mal nach der Zeile *GOSUB zeichnen* ein. Wenn Sie das Programm mit dem Demo-Programm von *IFFuniversal* laden, bekommen Sie von der Riesengrafik allerdings nur die linke obere Ecke zu sehen. Die kleine Demo-Routine kann das Bild nicht verschieben. Diese Grafik konnte aus Kapazitätsgründen nicht mit auf die Diskette.

16.3 Speichern im ACBM-Format

Diese Art der Datenspeicherung ist eine Mischung aus den beiden bisher besprochenen Systemen. Einerseits sind die Bilddaten in Chunks wie im IFF-ILBM-Standard abgelegt. Andererseits ist der BODY-Chunk, der das Bild selbst zeilen- und ebenenweise speichert, durch einen ABIT-Chunk (AmigaBITplanes) ersetzt worden. Dieser Chunk enthält hintereinander die kompletten BitPlanes, wie wir es vom GET-Datenfeld kennen.

Kapitel 17

Apfelmännchen & Co.

Fantastische Bilder lassen sich auch mit der grafischen Darstellung von mathematischen Formeln auf den Bildschirm zaubern. Die am häufigsten eingesetzten fraktalen Grafiken sind die *mandelbrotschen Grafiken*, so genannt nach ihrem Entdecker, dem Mathematiker Mandelbrot. Besser bekannt sind die Grafiken unter der Bezeichnung Apfelmännchen. Die typische Grundform der Grafik stellt solch ein Apfelmännchen dar. Die Berechnung der Mandelbrotmengen beruht auf den komplexen Zahlen. Dieser Bereich der Zahlensysteme ist sehr kompliziert. Um das Thema detailliert abzuhandeln, wären sehr umfangreiche Erklärungen notwendig, die den Rahmen eines Grafikbuches sprengen würden. Da dieses Buch auch von Computer-Anhängern gelesen werden soll, die die höhere Mathematik nicht kennengelernt haben, kann zum besseren Verständnis der Zusammenhänge ein kurzer Abriß nicht schaden.

17.1 Begriffe

Die Zahlen ordnet man ein unter den drei Oberbegriffen

- reelle Zahlen,
- imaginäre Zahlen und
- komplexe Zahlen.

Mit den reellen Zahlen werden Sie am häufigsten zu tun haben. Diese Zahlen werden noch weiter untergliedert. Da das für unsere Grafik ohne Belang ist, schenken wir uns die weitere Unterteilung. Mit dieser Vereinfachung können wir daher sagen, daß das System der reellen Zahlen sämtliche ganzen Zahlen, Brüche, Wurzeln und die transzendenten Zahlen (z.B. Pi,e) umfaßt.

Bei der imaginären Zahl gibt es den Begriff der imaginären Einheit: $i = +\sqrt{-1}$, so daß $i*i=-1$ ist. Eine imaginäre Zahl ist dann das Produkt aus der imaginären Einheit und einer reellen Zahl. Folgender Ausdruck stellt also eine imaginäre Zahl dar: $4*i=+\sqrt{-16}$.

Wird nun eine imaginäre Zahl mit einer reellen Zahl verbunden, so wird diese komplexe Zahl genannt. Bei dem Ausdruck z=a+i*b ist *z* die komplexe Zahl, *a* der Realteil und *b* deren Imaginärteil. Wenn man beachtet, daß i*i=–1 ist, so gelten für die komplexen Zahlen die gleichen mathematischen Regeln wie für die reellen Zahlen.

Bei der grafischen Darstellung der komplexen Zahlen (Gaußsche Zahlenebene) wird diese durch einen Punkt mit der Koordinate *a* für den Realteil (X-Achse) und der Koordinate *b* für den Imaginärteil (Y-Achse) dargestellt. In der Grafik steht dann jedes Pixel des Bildschirmes für eine komplexe Zahl. Die Berechnung erfolgt durch schrittweises Annähern an die Lösung. Dieses Verfahren nennt man *Iteration*. Dabei wird immer wieder derselbe Rechenschritt auf den zuvor errechneten Wert angewendet. Nach der Anzahl der Iterationen wird die Farbe des Grafikpunktes gesetzt.

Ein einfaches Beispiel, das nichts mit der grafischen Darstellung zu tun hat, soll Ihnen zeigen, wie bei der Iteration vorgegangen wird. Bei der Berechnung der Quadratwurzel durch Iteration wird als Ausgangswert die Zahl eingesetzt, aus der die Wurzel gezogen werden soll. Dieser Ausgangswert dient gleichzeitig als Startwert für die erste Näherungsrechnung. Die erste Iteration ergibt sich aus dem Startwert zuzüglich dem Ergebnis der Division von Startwert durch Ausgangswert. Das Ganze wird dann noch halbiert. Das Ergebnis der ersten Näherungsrechnung wird nun als neuer Startwert in die Formel eingesetzt. Bei jeder Berechnung (Iteration) nähert sich das Ergebnis dem richtigen Wert, bis es nicht mehr genauer geht. Der folgende Vierzeiler setzt die theoretischen Überlegungen in ein praktisches Beispiel um:

```
REM Wurzelberechnung  Pfad: Spezial/17Mandelbrot/Wurzel
'P17-1
start: INPUT "Bitte Zahl, Iterationstiefe eingeben: ",w,it%
PRINT "Berechnung mit SQR:   "SQR(w):x=w
'Iterationsformel: x(n+1)=(x(n)+w/x(n))/2
FOR i=1 TO it%:x=(x+w/x)/2: NEXT
PRINT "Ber. durch iteration: "x:PRINT :GOTO start
```

Am besten erkennen Sie die Annäherung an das gewünschte Ergebnis, wenn Sie die gleiche Zahl mit steigender Iterationstiefe eingeben. Sie werden feststellen, daß bei kleinen Zahlen bereits 4 bis 5 Iterationen zum richtigen Resultat führen.

17.2 Darstellung

Schauen wir uns nun an, wie die Grundlagen bei der Mandelbrot-Grafik umgesetzt werden. Die Grundformel wird nach dem realen und Imaginärteil aufgelöst. Mann beginnt mit der Zahl Null und zieht davon die komplexe Zahl ab. Das Ergebnis nimmt man ^2 und zieht wieder die komplexe Zahl ab. Das Ergebnis wird wieder quadriert, die komplexe Zahl abgezogen und so weiter. Prinzipiell geht man von einem fixen Punkt der X- und Y-Ebene aus. Die verschiedenen Werte der Parameter der komplexen Zahl während der Iteration werden Pixel für Pixel zu einer Grafik zusammengesetzt. (Dreht man die Betrachtungsweise herum und geht von den Parametern der komplexen Zahl aus, so erhält man die Juliamenge. Die Berechnungen sind dabei aber noch zeitaufwendiger.) Der darzustellende Bereich der Zahlenebene wir durch die Variablen *Xmin* und *Xmax* für die X-Achse und durch die Veränderlichen *Ymin* und *Ymax* für die Y-Achse vorgegeben. Die Schrittweite ergibt sich aus dem ausgewählten Zahlenbereich dividiert durch die Bildhöhe bzw. die Bildbreite:

```
xd=(xmax-xmin)/Fensterbreite%    'Schrittweite X-Richtung
yd=(ymax-ymin)/Fensterhöhe%      'Schrittweite Y-Richtung
```

Die Ergebnisse der Berechnungen sehen sehr merkwürdig aus. Bei einigen Werten werden sie immer kleiner, bei anderen steigen sie schnell an, und wieder andere pendeln unschlüssig hin und her. In der Grundform ergibt sich dadurch die Grafik des Apfelmännchens. Dabei ist besonders die Randzone des Männchens interessant. Auf ihm sitzen weitere Apfelmännchen, auf denen wieder solche sitzen. Man spricht dabei von Selbstähnlichkeit. Aber auch bizarre Formen lassen sich speziell in der Übergangszone finden. Ihrer Experimentierfreudigkeit sind da keine Grenzen gesetzt.

Überschreiten die Werte eine bestimmte Grenze (Iterationsschranke), in unserem Programmbeispiel den Wert 4, so wird die Berechnung abgebrochen. In diesem Fall wird ein farbiger Punkt gesetzt. Führt die Berechnung innerhalb der eingegebenen Iterationen zu keiner Überschreitung des Grenzwertes, so wird kein Punkt gesetzt, es bleibt also die Hintergrundfarbe 0.

Die Farbgebung richtet sich, wie bereits erwähnt, nach der Anzahl der Iterationen. Dabei wird diese Anzahl modular durch 15 dividiert und das Ergebnis um 1 erhöht. Dadurch erhalten wir die oberen 15 Farbwerte, die zum Setzen der Punktfarbe benötigt werden.

Den vollständigen Berechnungs-Algorithmus entnehmen Sie bitte dem folgenden Bild:

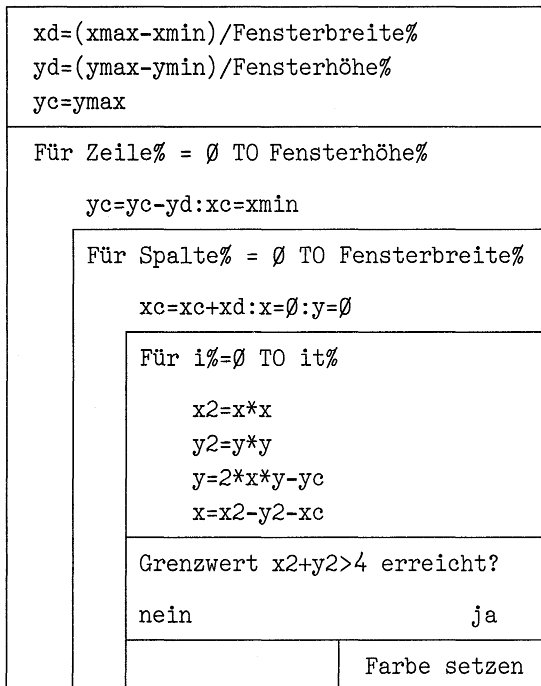


Bild 17.1:
Struktogramm der Berechnung

17.3 Das Programm

Das Hauptproblem der fraktalen Grafik unter Basic liegt in der Dauer der notwendigen Berechnungen. Berücksichtigen wir eine Iterationstiefe von 300 bei einer Fensterbreite von 631 Pixel und einer Fensterhöhe von 242 Bildpunkten, so ergeben sich dafür

```
3ØØ*631*242 = 45.81Ø.6ØØ Schleifendurchläufe.
```

Zwar durchlaufen natürlich nicht alle Bildpunkte die maximale Anzahl von Iterationen, aber trotzdem benötigt jedes Grafikbild eine ganze Menge an Zeit. Je genauer man das Bild darstellen will, je mehr Iterationen man also durchführt, um so höher ist der Zeitaufwand. Um den schnellsten Algorithmus zu finden, habe ich als Testbild das Standard-Apfelmännchen mit folgenden Werten in etwa 50 Testläufen laufen lassen:

```
Bildschirmbreite: 32Ø
Bildschirmhöhe:   244
größtes X:        2.1
kleinstes X:     -Ø.7
größtes Y:        1
kleinstes Y:     -1
Iterationen:      1Ø
```

Die folgende Programmversion benötigt für dieses grobe Bild (zu wenig Iterationen) immerhin schon 64 Minuten. An dem Faktor Zeit kommt man also nicht vorbei. Daher wurde versucht, im Programm die anderen Funktionen zu beschleunigen. So kann ein Testbild mit 1/4 Breite und Höhe in sehr kurzer Zeit erstellt werden, um Zeichenversuche mit ungünstigen Werten zu minimieren. Zur Speicherung und zum Einlesen fertiger Bilder wird das schnelle Programm *IFFuniversal* aus dem Kapitel für die Grafikspeicherung eingesetzt. Außerdem haben Sie die freie Auswahl zwischen 6 verschiedenen Farbtafeln. Eine weitere Besonderheit ist die aufrufbare Farbrotation, die fantastische Effekte hervorruft. Betrachten wir uns jedoch zuerst das Programm im einzelnen.

```
REM Apfel  Pfad: Spezial/17Mandelbrot/Apfel
'P17-1
'by H-R Henning
'schnellste Version test-Bild in 64 min. mit PSET
'breite 320 * x 2.1/-.7 * y 1/-1 * 10 Iterationen

initMenue:
  DIM f%(15):DIM Bild&(1226)
  GOSUB libraryauf        'im Programm IFFuniversal
  SCREEN 1,320,256,4,1
  WINDOW 2,tit-,(0,0)-(310,160),0,1
  filename-="Mini":GOSUB einlesen
  GET (0,10)-(80,69),Bild&

Schirm:
  SCREEN 1,320,256,4,1
  v&=ViewAddress& :vp&=PEEKL(v&)
  tit-="Apfelmännchen"
  ti-=tit-+CHR-(0)
  WINDOW 2,tit-,(0,20)-(310,160),0,1
  FOR j=0 TO 10
    FOR i=j*9 TO 9*j+9
      PUT (i,3),Bild&,XOR
      PUT (i,3),Bild&,XOR
    NEXT i
    ti-=" "+ti-
    CALL SetWindowTitles&(WINDOW(7),SADD(ti-),SADD(ti-))
  NEXT j
  PUT (i,3),Bild&,PSET
```

```
start:
  LOCATE 11,12:PRINT "F1 = Bild laden"
  LOCATE 12,12:PRINT "F2 = Bild zeichnen"
  LOCATE 13,12:PRINT "F3 = E N D E"
  frage:ta-=INKEY-:IF ta-="" THEN frage
  IF ta-<CHR-(129) OR ta-> CHR-(131) THEN frage
  verz%=ASC(ta-)-128
  ON verz% GOTO laden,zeichnen,Ende2
  BEEP: GOTO start

zeichnen:
  CLS
  GOSUB namelesen
  LOCATE 12,1:PRINT "lores (32Ø * 256 Bildpunkte) = F4"
  LOCATE 13,1:PRINT "hires (64Ø * 256 Bildpunkte) = F5"
  frage2:ta-=INKEY-:IF ta-="" THEN frage2
  IF ta-<CHR-(132) OR ta-> CHR-(133) THEN frage2
  yf%=243
  WINDOW CLOSE 2:SCREEN CLOSE 1
  IF ta-=CHR-(132) THEN
    xf%=319
    SCREEN 1,32Ø,256,4,1
  ELSE
    xf%=639
    SCREEN 1,64Ø,256,4,2
  END IF
  WINDOW 2,filename-,(Ø,1ØØ)-(31Ø,16Ø),Ø,1
  v&=ViewAddress& :vp&=PEEKL(v&)
  rp&=vp&+4Ø :fscr&=vp&-44
  GOSUB farben
```

Das Programm öffnet keine eigenen Library-Routinen. Mit der Verzweigung GOSUB »libraryauf« werden die Routinen im Programm *IFFuniversal* aktiviert. Ein entsprechender Zeiger in diesem Programm sorgt dafür, daß die Routine kein zweites Mal angesprungen wird. Es folgt der Aufbau des Menüs. Dafür wird ein kleines Apfelmännchen eingelesen. Zusammen mit einem Lauftext gibt es dem Menü etwas Pep. Über die Funktionstasten F1 bis F3 können die Programmfunktionen *Bild laden*, *Bild zeichnen* und *ENDE* aufgerufen werden. Wird F2 für *Bild zeichnen* aufgerufen, so wird über die Funktionstasten F4 und F5 die Wahlmöglichkeit zwischen hoch- oder niedrigauflösender Darstellung gegeben. Die Farbtiefe ist für beide Varianten einheitlich auf 4 für 16 Farben festgelegt. Entsprechend der gewählten Einstellung wird der Bildschirm und ein Fenster geöffnet. Aus der Subroutine »farben« wird die erste Farbtafel aufgerufen.

```
eingeben:
  INPUT "größtes X:   ",xmax
  INPUT "kleinstes X: ",xmin
  INPUT "größtes Y:   ",ymax
  INPUT "kleinstes Y: ",ymin
  INPUT "Iterationen: ",it%
  INPUT "Testlauf j/n ",jn-
  WINDOW CLOSE 2
  zd%=12
  IF jn-="j" THEN
    xf%=xf%/4: yf%=yf%/4
    WINDOW 2,,(Ø,Ø)-(Ø+xf%,Ø+yf%),Ø,1
  END IF
  CLS
Berechnung:
  xd=(xmax-xmin)/xf%    'Schrittweite X-Richtung
  yd=(ymax-ymin)/yf%    'Schrittweite Y-Richtung
  yc=ymax
  FOR z% = zd% TO yf%+zd%
    yc=yc-yd:xc=xmin
    FOR s% = Ø TO xf%
      xc=xc+xd
      x=Ø:y=Ø
      FOR i%=Ø TO it%
        x2=x*x
        y2=y*y
        y=2*x*y-yc
        x=x2-y2-xc
        IF x2+y2>4 THEN col
      NEXT i%
      weiter:
    NEXT s%
  NEXT z%
```

Nun werden die einzelnen Parameter für die Grafik eingegeben. Wird ein Testlauf mit verkleinerter Grafik gewünscht, so werden die Variablen für die Bildbreite *xf%* und für die Höhe der Grafik *yf%* auf ein Viertel reduziert. Es folgt die Berechnung der einzelnen Pixel der Grafik wie bereits ausführlich erläutert.

```
  FOR n= 1 TO 1Ø
    BEEP
    FOR t=1 TO 1ØØØ:NEXT
  NEXT
bildzeigen:
  v&=ViewAddress& :vp&=PEEKL(v&)
  rp&=vp&+4Ø
  tit-="F1=wechsel F2=rotieren F3=speich F4=Menu"+CHR-(Ø)
  ti&=SADD(tit-)
  CALL Move&(rp&,1,8)
  CALL SetDrMd&(rp&,2)
  CALL Text&(rp&,ti&,4Ø)
  tast: tt-=INKEY-:IF tt-="" THEN tast
  CALL Move&(rp&,1,8)
  CALL Text&(rp&,ti&,4Ø)
  IF tt-<CHR-(129) OR tt-> CHR-(131) THEN Men
  verz%=ASC(tt-)-128
  ON verz% GOSUB farben,farbrotieren,speichern
  GOTO bildzeigen

Ende2:
  LIBRARY CLOSE:WINDOW CLOSE 2:SCREEN CLOSE 1
  ERASE Bild&:END
```

Sobald die Grafik fertiggestellt ist, ertönen 10 Pieptöne. Da anzunehmen ist, daß niemand einige Stunden vor dem Amiga sitzen bleibt, um das Entstehen der Grafik zu bewundern, soll der Anwender dadurch ans Gerät zurückgerufen werden. Denn jetzt gibt es etwas zu tun. Durch die Library-Routinen *Move*, *SetDrMd* und *Text* werden über die Titelleiste des Fensters 4 Wahlmöglichkeiten offeriert:

Mit *F1=wechsel* können nacheinander sämtliche 6 Farbtafeln abgerufen werden. Sie können so entscheiden, welche Farbkombination Sie abspeichern wollen.

Die zweite Funktion *F2=rotieren* bietet ein besonderes optisches Schmankerl. Die Farben 1 bis 15 rotieren, das heißt, sie werden abwärtsfallend untereinander ausgetauscht. Lassen Sie sich von den reizvollen Effekten überraschen.

Über die Funktionstaste F3 leiten Sie das Abspeichern des Bildes ein. Sie springen dabei direkt in das Programm *IFFuniversal.*

Mit *F4=Menu* schließlich gelangen Sie wieder ins Start-Menü. Bei dem Label »ende2« werden Fenster, Screen und Library geschlossen. Damit ist das Hauptprogramm beendet, und wir sehen uns noch kurz die einzelnen Routinen an.

```
Men:WINDOW CLOSE 2:SCREEN CLOSE 1:GOTO Schirm

col:
  fa% =i% MOD 15+1
  CALL SetAPen&(rp&,fa%)
  CALL WritePixel&(rp&,s%,z%)
  GOTO weiter

farben:
  FOR i%=Ø TO 15:READ f%(i%):NEXT
  READ zeig: IF zeig THEN RESTORE Farbwerte
  CALL LoadRGB4&(vp&,VARPTR(f%(Ø)),16)
RETURN

farbrotieren:
  fs%=f%(1)
  FOR i%=1 TO 14
    f%(i%)=f%(i%+1)
  NEXT
  f%(15)=fs%
  CALL LoadRGB4&(vp&,VARPTR(f%(Ø)),16)
  ts-= INKEY-:IF ts-="" THEN farbrotieren
RETURN

laden:
  bm&=Ø:GOSUB namelesen:WINDOW CLOSE 2
  GOSUB einlesen
  IF bm&=Ø THEN SCREEN CLOSE 1:GOTO schirm
  GOTO bildzeigen
```

Jedesmal wenn die Subroutine »farben« angesprungen wird, wird eine neue Farbtafel in das Feld *f%()* eingelesen. Wie bereits erwähnt, stehen 6 Farbtafeln zur Verfügung. Sie können die Auswahl jederzeit erweitern, wenn Sie das folgende beachten. Das Datenfeld besteht aus 16 Farben. Für jede Farbe ist ein hexadezimales Datenwort in Wortlänge (2 Byte) zu finden. Das erste Nibbel (Halbbyte) ist Null, das zweite enthält den Rotanteil, das dritte den Grünanteil und das vierte den Blauanteil der Farbe. Dabei ist 15 ($f) der höchste und 0 der niedrigste Wert. Die Anfangsadresse dieses Datenfeldes *f%(0)* wird an die Bibliotheks-Routine *LoadRGB4&* übergeben. Damit steht die neue Farbtafel zur Verfügung. Am Ende eines jeden Datenfeldes finden Sie eine zusätzliche Zahl 0 oder –1. Sie wird mit der Variablen *zeig* eingelesen und dient zur Kennung, ob noch ein weiterer Datensatz folgt. Wird *zeig* logisch wahr (–1), so ist der letzte Satz eingelesen und der Datenzeiger wird mit RESTORE wieder an den Anfang des ersten Datenfeldes gesetzt.

Die Farbrotation bei dem Label »farbrotieren« ist eigentlich eine recht einfache Angelegenheit. Die Farbwerte der Farbe 1 werden in der Variablen *fs%* gesichert. In einer Schleife erhalten dann die Farben 1 bis 14 die Farbwerte der nächsthöheren Farbe. Die Farbe 15 erhält als letztes die in *fs%* gesicherten Farbanteile. Die Bibliotheks-Routine *LoadRGB4&* hinterlegt die neue Farbtafel. Die Farben rotieren, bis eine beliebige Taste gedrückt wird. Der Farbwechsel geht sehr schnell vonstatten. Wenn es Ihnen zu schnell wird, können Sie an dieser Stelle eine Zeitschleife einbauen.

```
Farbwerte:
DATA &h0000
DATA &h0700,&h0900,&h0b00,&h0d00,&h0f00
DATA &h0050,&h0070,&h0090,&h00b0,&h00d0
DATA &h0005,&h0007,&h0009,&h000b,&h000d,0

DATA &h0000
DATA &h0600,&h0700,&h0800,&h0900,&h0a00
DATA &h0b00,&h0c00,&h0d00,&h0e00,&h0f00
DATA &h0f08,&h0f0a,&h0f0c,&h0f0d,&h0f0f,0

DATA &h0000
DATA &h0800,&h0900,&h0a00,&h0b00,&h0c00
DATA &h0080,&h0090,&h00a0,&h00b0,&h00c0
DATA &h0008,&h0009,&h000a,&h000b,&h000c,0

DATA &h0000
DATA &h0060,&h0070,&h0080,&h0090,&h00a0
DATA &h00b0,&h00d0,&h00f0,&h00f6,&h00f9
DATA &h00fb,&h00fd,&h00ff,&h08ff,&h0bff,0

DATA &h0000
DATA &h0ffd,&h0ffb,&h0ff9,&h0ff0,&h0fe0
DATA &h0fd0,&h0fc0,&h0fb0,&h0fa0,&h0f90
DATA &h0e90,&h0d90,&h0c90,&h0c97,&h0d97,0

DATA &h0000
DATA &h000a,&h000b,&h000c,&h000d,&h000e
DATA &h000f,&h004f,&h005f,&h006f,&h007f
DATA &h008f,&h009f,&h00af,&h00bf,&h00cf,-1
```

Zum Laden und Abspeichern der Grafiken greifen wir auf das Programm *IFFuniversal* zurück. Hängen Sie es bitte mit MERGE an das Hauptprogramm an. Zur Anpassung an das Apfelmännchen muß eine Zeile außer Kraft gesetzt werden. Die Änderung entnehmen Sie bitte dem folgenden Programmausschnitt.

```
##############################
'Bitte folgendes Programm anfügen
REM IFFuniversal
REM V4.11
'##############################
'bitte setzen Sie vom Programm IFFuniversal, zwischen
'den beiden Labels schluss und schluss2 die Zeile
'mit der Tastaturabfrage außer Kraft
  schluss:
    'tast1:ta-=INKEY-:IF ta-="" THEN tast1  ##########
  schluss2:
'##########################
```

Zum Schluß des Programmes noch ein paar Worte zum benötigten Speicherbedarf für die einzelnen Grafiken auf der Diskette. Wie Sie gelesen haben, stehen zwei verschiedene Auflösungen zur Wahl. Bei niedriger Auflösung (lores) benötigen Sie 320/8 Byte für die Bildbreite * 256 Zeilen * Tiefe 4 = 40960 Byte, zuzüglich 116 Byte für das Icon macht 41076 Byte. Bei hoher Auflösung benötigen Sie sogar 82 Kbyte für eine einzige Grafik. Achten Sie bitte darauf, daß auf Ihrer Diskette genügend Platz vorhanden ist.

17.4 Ein paar Beispiele

Auf der Diskette zu diesem Buch finden Sie bereits 2 fertige Grafiken vor. Sicherlich interessieren Sie die zugehörigen Parameter und die benötigte Zeit zur Erstellung der Grafiken.

frak1

Die Grafik zeigt die Grundform des Apfelmännchens. Auf der Diskette werden 41076 Byte Speicherkapazität benötigt. Es dauerte 11 Stunden und 11 Minuten, bis das Programm die Grafik fertig hatte. Es ist durchaus möglich, daß weniger Iterationen (= weniger Zeitaufwand) das gleiche Ergebnis bringen. – Die Höhe des Bildes beträgt, wie bei den anderen Grafiken auch, 256 Zeilen. Ebenso ist die Tiefe 4 fest vorgegeben. Die anderen Parameter der Grafik lauteten:

```
Bildschirmbreite: 320
größtes X:         2.1
kleinstes X:      -0.7
größtes Y:         1
kleinstes Y:      -1
Iterationen:       300
```

frak2

Diese Grafik zeigt einen Ausschnitt aus der Grundform des Apfelmännchens. Der kleine Apfel halblinks an der oberen Randzone des großen Apfels wurde herausgezogen. Besonders gut kann man die Selbstähnlichkeit der Grafik erkennen. Sogar auf dem kleinen Apfel sitzen weitere Äpfel, auf denen sich wieder Äpfel befinden ...

Die Grafik wurde in knapp vier Stunden, genau in 3 Stunden und 55 Minuten, gezeichnet. Wie beim ersten Beispiel wurden 41076 Bytes Speicherkapazität benötigt. Auch die fest eingestellten Werte für die Bildhöhe und Tiefe sind gleich. Die anderen Parameter wurden wie folgt eingegeben:

```
Bildschirmbreite: 32Ø
größtes X:       -Ø.2
kleinstes X:     -Ø.45
größtes Y:        Ø.7
kleinstes Y:      Ø.5
Iterationen:      1ØØ
```

Kapitel 18

Der Copper

Der Copper ist eines der wichtigsten Bestandteile der Systemsoftware. Auch bei der Basic-Programmierung stoßen wir immer wieder auf diesen Namen. Der Name selbst ist wahrscheinlich eine etwas ungewöhnliche Abkürzung für Coprozessor. Er ist ein Teil vom Agnus, einem der drei Spezialchips des Amiga. Er ist letztendlich für die Kontrolle der Grafik-Ausgabe zuständig. Durch den direkten Speicherzugriff über DMA (Direct Memory Access) und da er seine Aufgaben ohne den Prozessor erledigt, ist er ein echter Coprozessor.

Bei unserem Monitor flitzt der Elektronenstrahl von der linken oberen Ecke zeilenweise nach unten und springt dann wieder in die linke obere Ecke zurück. Das Ganze passiert 50 Mal in einer Sekunde. Dem Copper verbleibt also genügend Zeit, um bei dieser Abwärtsbewegung des Elektronenstrahles zum Beispiel 8 Sprites nebeneinanderzusetzen und nachzusehen, welche Farben der Anwender sich dafür ausgesucht hat. Mit einer Zeile Abstand darunter kann er die nächste Reihe Sprites setzen etc. Nebenbei kontrolliert er noch den Blitter und das allgemeine Farbregister.

18.1 Anwender-Copperlisten

Was genau er alles auf dem Weg des Elektronenstrahles zu erledigen hat, wird in Listen festgehalten. Diese CopperLists des Systems sollen uns hier nicht interessieren. Viel aufregender ist, daß auch der Anwender solche Listen programmieren kann. Diese Listen nennt man UserCopperLists. Sie bestehen aus einer Reihenfolge von Befehlen, die der Copper der Reihe nach ausführt. Ist die Liste durchgearbeitet, fängt er wieder von vorne an.

Für diese Aufgaben hat der Copper sogar eigene Befehle. Der WAIT-Befehl wartet, wie schon der Name sagt, bis eine vorgegebene Position auf dem Bildschirm vom Elektronenstrahl überstrichen wird. Dabei gilt für X ein Bereich von 0 bis 226. Das reicht natürlich für keinen View-Modus aus. Deshalb wird bei einem lores-Schirm von 320 Pixel Breite in Schritten von 4 Pixel und bei einem Bildschirm von 640 Bildpunkten Breite in 8er-Schritten vorgegangen. Mit einem X-Wert von 356 teilt man dem

Copper mit, daß die Liste abgearbeitet ist und daß er gefälligst wieder von vorne anzufangen hat. Die Y-Richtung kann von 0 bis 255 programmiert werden.

Der zweite Befehl MOVE verschiebt Daten in ein Zielregister. Zuerst kommt also der WAIT-Befehl, der wartet, bis eine bestimmte Position auf dem Bildschirm erreicht ist, und dann der MOVE-Befehl, der einen Befehl ausführt. Es dürfen auch mehrere MOVE-Befehle nach einem WAIT in die CopperLists geschrieben werden. Diese Befehle werden durch Routinen der Grafik-Library aufgerufen.

18.2 Die Werkzeuge

Das fortwährende Abarbeiten der Befehle in den Listen ist eine feine Sache. Sie können am Anfang eines Programmes die Listen initialisieren und sie dann vergessen. Ohne Warteschleifen oder anderen Schnickschnack vollzieht der Copper seine Aufgabe völlig selbständig. Sie können also ganz normal in Ihren Programmen fortfahren. Da ich Ihnen inzwischen den Mund wässerig gemacht habe, schauen Sie sich dazu erst einmal ein kleines Beispiel an.

```
REM Flagge  Pfad: Spezial/18Copper/Flagge
'P18-1
CLEAR
DEFINT a-z
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):tiefe=1 '!!!
GOSUB LibraryOeffnen
GOSUB Speicher      :IF Fehl THEN ende
SCREEN 1,64Ø,sh,tiefe,2:WINDOW 2,,,Ø,1
vp&=PEEKL(WINDOW(7)+46)+44
GOSUB UCopperListeAnlegen
'UserCopperListe in ViewPort schreiben
POKEL vp&+2Ø,uc&
CALL RethinkDisplay&

WHILE INKEY-="":WEND

ende:
  WINDOW CLOSE 2:SCREEN CLOSE 1
  FOR i=1 TO 1ØØØ:NEXT
  LIBRARY CLOSE
  IF Fehl THEN BEEP:PRINT "Speicher nicht ausreichend"
END
```

```
LibraryOeffnen:
  DECLARE FUNCTION AllocMem&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
  LIBRARY ":bue/exec.library"
RETURN

Speicher:
  art&=3+(2^16)
  uc&=AllocMem&(12,art&) 'fuer UserCopperList
  IF uc&=Ø THEN Fehl=2
RETURN

UCopperListeAnlegen:
  a1=1:a2=sh/3:a3=sh/3*2
  '1/3 schwarz
  CALL CWait&(uc&,a1,1Ø)    :CALL CBump&(uc&)
  CALL CMove&(uc&,384,&HØ):CALL CBump&(uc&)
  '1/3 rot
  CALL CWait&(uc&,a2,1Ø)    :CALL CBump&(uc&)
  CALL CMove&(uc&,384,&HFØØ):CALL CBump&(uc&)
  '1/3 gelb
  CALL CWait&(uc&,a3,1Ø)    :CALL CBump&(uc&)
  CALL CMove&(uc&,384,&HFFØ):CALL CBump&(uc&)
  'Ende der CopperList
  CALL CWait&(uc&,1ØØØ,256):CALL CBump&(uc&)
RETURN
```

Habe ich zuviel versprochen? Beachten Sie bitte die Tiefe des Screens. Er besitzt nur eine BitPlane! Aber es kommt noch besser. Bevor ich diese Behauptung durch ein weiteres Beispiel belegen kann, schauen wir uns erst einmal das aktuelle Programm an. An Hand dieses Beispiels werden wir die Technik der Copper-Programmierung detailliert besprechen. Wir benötigen dazu drei Libraries, die im einzelnen für folgende Routinen eingesetzt werden:

Intuition	RethinkDisplay, zum Überdenken des ViewPorts
Grafik	Für die Copper-Befehle
Exec	AllocMem, zur Speicherreservierung.

Speziell die letzte Library wollte ich einsparen. Schließlich bietet die Intuition das Pendant *AllocRemember.* Mit einem etwas anderen Programmaufbau funktionierte das Programm sogar ohne eigenen Screen und ohne eigenes Fenster (das wir wieder nur zur Kommunikation zwischen Anwender und Programm benötigen). Leider klaute das Programm bei jedem Aufruf ein Stückchen vom System-Speicher. Warum das so ist, werden Sie etwas später erfahren.

Es werden ein hires-Screen mit der Bitmap-Tiefe 1 und ein Fenster eingerichtet. Nun wird es interessant. In der Subroutine »UCopperListeAnlegen« kommen wir bereits zum Kern der Sache. Der Bildschirm soll in drei verschiedenen Hintergrundfarben dargestellt werden. Dazu teilen wir die Screen-Höhe in drei gleiche Abschnitte. Der erste Abschnitt soll schwarz gezeichnet werden. Mit der Grafik-Routine

---

Format: `CWAIT(c,v,h)`

---

schreiben wir für den Copper eine Instruktion in die Liste, die besagt, daß er auf die vertikale *v* und horizontale *h* Position des Elektronenstrahls zu warten hat. In unserem Programm hat *h*, also die X-Position, den Wert 10. Damit findet der Wechsel von einer zur anderen Farbe im nicht sichtbaren Bereich statt. Bis zu einem Wert von etwa 50 ist der Farbwechsel nicht zu sehen. Größere Werte sind auch kein Beinbruch, solange Sie nicht den Maximalwert überschreiten und damit die Liste beenden. Die Position *h* für den Y-Wert legen wir auf die erste Grafikzeile. Setzen Sie dafür einen negativen Wert, zum Beispiel –10, ein, so wird der Bereich bis zur Grafikzeile 0 schwarz gefärbt, und danach erscheint die übliche blaue Hintergrundfarbe. Die zweite Grafik-Routine

---

Format: `CBump(c)`

---

veranlaßt, daß in der Liste eine Position weiter gegangen wird. Der nächste Befehl soll ja an die richtige Stelle geschrieben werden. Kommen wir gleich zur letzten Grafik-Routine, die wir zur Copper-Programmierung benötigen.

Format: `CMOVE(c,a,v)`

Die Routine schreibt als Anweisung in die Copper-Liste, daß der Inhalt von *v* in das Hardware-Register *a* kopiert werden soll. Dazu ist nicht die komplette Speicherstelle des Hardware-Registers notwendig, sondern nur der Offset vom Beginn des Hardware-Registers. In unseren Programmen setzen wir nur die Farbregister ein, die nachstehend aufgeführt sind. Die Register-Adressen sind als Offset zum Beginn des Hardware-Registers anzusehen.

Bezeichnung	**Offset**
Farbe 0	$180 (384)
Farbe 1	$182 (386)
Farbe 2	$184 (388)
Farbe 3	$186 (390)
...	
Farbe 28	$1b8 (440)
Farbe 29	$1ba (442)
Farbe 30	$1bc (444)
Farbe 31	$1be (446)

Kommen wir auf unser Programm zurück. Nacheinander werden die einzelnen Routinen für den schwarzen, roten und gelben Bildschirmbereich aufgerufen. Die letzten beiden Routinen beenden die UserCopperList. Sie entsprechen dem Makro CEND, das genau die gleichen Routinen aufruft. Mit einem Wert von 256 für die X- Position *h* wird dem Copper mitgeteilt, daß die Liste beendet ist und daß die Arbeit wieder von vorne beginnt. Der Wert von 1000 für die Y-Position ist nur symbolisch zu sehen. Der Wert wird nicht berücksichtigt.

Wieder ins Hauptprogramm zurückgekehrt, wird die Struktur UCopperList in die Position 20 der ViewPort-Struktur geschrieben. Mit *RethinkDisplay* übernimmt die Intuition wieder die Abstimmung mit den anderen Grafik-Ausgabeelementen. Sie haben sich bestimmt schon gewundert, warum bei den Beschreibungen der einzelnen Routinen die Variable *uc&* für die Struktur UserCopperList unerwähnt geblieben ist. Da diese in einen größeren Zusammenhang gehört, folgt die Erklärung erst jetzt.

Bei der Speicherreservierung haben wir 12 Byte für die Struktur reserviert. Beim Eintragen der einzelnen Copper-Befehle wurde eine Liste in der Länge der Anzahl der Befehle erstellt. Die Adresse der Liste wird in die Struktur eingetragen. Deshalb haben wir bei jeder Routine die Struktur angegeben. Wenn wir die User-Liste nicht mehr haben wollen, wissen wir daher nicht, wieviel Speicherplatz freigegeben werden muß. Unter Intuition gibt es dafür nur eine Methode, der Screen muß geschlossen werden. Damit wird die Copperliste aus dem ViewPort genommen, der ViewPort, die Copper-Liste und die Struktur etc. werden mit ihren Speicherbereichen an das System zurückgegeben. Deshalb finden Sie in diesem Programm ausnahmsweise keine Freigabe des reservierten Speichers mit *FreeMem*.

18.3 Die Anwendung

Nach soviel Theorie haben Sie sich etwas Entspannung verdient. Sehen Sie an unserem nächsten Beispiel, was der Copper mit der Farbe alles anstellen kann.

```
REM color5ØØ  Pfad: Spezial/18Copper/color5ØØ
'P18-2
CLEAR
DEFINT a-z
z=15: DIM x2(z),y2(z),x3(z),y3(z)
sh=PEEKW(PEEKL(WINDOW(7)+46)+14):tiefe=1 '!!!
GOSUB LibraryOeffnen
'erster Durchgang
  GOSUB Speicher      :IF Fehl THEN ende
  SCREEN 1,32Ø,sh,tiefe,1:WINDOW 2,,,Ø,1
  scr&=PEEKL(WINDOW(7)+46):rp&=scr&+84
  vp&=scr&+44:WINDOW CLOSE 2
  GOSUB UCopperListeAnlegen1
  'UserCopperListe in ViewPort schreiben
    POKEL vp&+2Ø,uc&
    CALL rethinkDisplay&
  GOSUB TextAusgeben
  ti#=TIMER:WHILE TIMER<ti#+2:WEND
  SCREEN CLOSE 1
  PRINT "einen kleinen Augenblick ..."
  FOR i=1 TO 3ØØØ:NEXT :CLS

'zweiter Durchgang
  GOSUB Speicher      :IF Fehl THEN ende
  SCREEN 1,64Ø,sh,tiefe,2:WINDOW 3,,,Ø,1
  vp&=PEEKL(WINDOW(7)+46)+44
  GOSUB UCopperListeAnlegen
  'UserCopperListe in ViewPort schreiben
    POKEL vp&+2Ø,uc&
    CALL rethinkDisplay&
  GOSUB zeichnen
ende:
  WINDOW CLOSE 3:SCREEN CLOSE 1
  FOR i=1 TO 1ØØØ:NEXT
  LIBRARY CLOSE
  IF Fehl THEN BEEP:PRINT "Speicher nicht ausreichend"
END
```

```
LibraryOeffnen:
  DECLARE FUNCTION AllocMem&() LIBRARY
  LIBRARY ":bue/intuition.library"
  LIBRARY ":bue/graphics.library"
  LIBRARY ":bue/exec.library"
RETURN

Speicher:
  art&=3+(2^16)
  uc&=AllocMem&(12,art&) 'fuer UserCopperList
  IF uc&=0 THEN Fehl=2
RETURN

UCopperListeAnlegen1:
  fa=4096
  FOR x=1 TO sh-2
    fa=fa-1
    CALL CWait&(uc&,x,10)    :CALL CBump&(uc&)
    CALL CMove&(uc&,384,fa) :CALL CBump&(uc&)
  NEXT x
  'Ende der CopperList
  CALL CWait&(uc&,10000,256):CALL CBump&(uc&)
RETURN

UCopperListeAnlegen:
  fa=4096:fa1=2048
  FOR x=1 TO sh-2
    fa=fa-1:fa1=fa1+1
    CALL CWait&(uc&,x,10)    :CALL CBump&(uc&)
    CALL CMove&(uc&,384,fa) :CALL CBump&(uc&)
    CALL CMove&(uc&,386,fa1):CALL CBump&(uc&)
  NEXT x
  'Ende der CopperList
  CALL CWait&(uc&,10000,256):CALL CBump&(uc&)
RETURN

zeichnen:
  xx=WINDOW(2)-2:yy=WINDOW(3)-2
  a=-1:x1=xx/2:x5=x1+20:x4=xx/4:y1=yy/2:y4=yy/4:y8=yy/8
  WHILE a
    CLS:b=24
    WHILE b
      LOCATE b,1:PRINT txt-(3)
```

```
      FOR i= Ø TO y1
        LINE (x5-2*i,y1-i)-(x5+2*i,y1+i),1,bf
      NEXT
      FOR i= y1 TO Ø STEP-1
        LINE (x5-2*i,y1-i)-(x5+2*i,y1+i),Ø,b
        LINE (x5-2*i-1,y1-i)-(x5+2*i+1,y1+i),Ø,b
      NEXT
      b=b-4:taste:CLS
    WEND
    CLS:b=8Ø:IF a=Ø THEN aus
    WHILE b
      FOR eck= Ø TO 12:AREA(RND*xx,RND*yy):NEXT
      CLS:AREAFILL
      b=b-1:taste
    WEND
    CLS:b=8Ø:IF a=Ø THEN aus
    WHILE b
      FOR i=Ø TO z
        LINE (x2(i),y2(i))-(x3(i),y3(i)),Ø
        x2(i)=ABS(x2(ia)+RND*x1-x4)
        IF x2(i)>xx THEN x2(i)=xmax
        y2(i)=ABS(y2(ia)+RND*y4-y8)
        IF y2(i)>yy THEN y2(i)=ymax
        x3(i)=ABS(x3(ia)+RND*x1-x4)
        IF x3(i)>xx THEN x3(i)=xmax
        y3(i)=ABS(y3(ia)+RND*y4-y8)
        IF y3(i)>yy THEN y3(i)=ymax
        LINE (x2(i),y2(i))-(x3(i),y3(i)),1
        ia=i
      NEXT
      b=b-1:taste
    WEND
    aus:
  WEND
RETURN

TextAusgeben:
  CALL SetRGB4&(vp&,1,Ø,Ø,Ø)
  CALL SetDrMd&(rp&,4)
  CALL SetAPen&(rp&,Ø)
  txt-(Ø)="Der Copper macht es moeglich"
```

```
  txt-(1)="5ØØ Farben auf einen Streich"
  txt-(2)="und das mit einer Bit-Ebene!"
  txt-(3)="Taste=ENDE"
  FOR j = 2Ø TO sh-4Ø
    CALL SetRast&(rp&,1)
    FOR i = Ø TO 2
      CALL Move&(rp&,44,i*16+j)
      CALL Text&(rp&,SADD(txt-(i)),LEN(txt-(i)))
    NEXT
    FOR n= 1 TO 5ØØ:NEXT
  NEXT
RETURN

SUB taste STATIC
  SHARED a%,b%
  IF INKEY-<>"" THEN a%=Ø:b%=Ø
END SUB
```

Wenn das keine Farbenpracht ist!?, dazu aus einer einzigen Bit-Ebene. Wenn es Ihnen nicht zu bunt wird, können Sie ja noch innerhalb der Grafikzeilen die Farben wechseln. Es ist überhaupt kein Problem, als Hintergrundfarbe(n) alle 4096 Farbtöne des Amiga darzustellen. – Um im Programm die unterlegte Schrift besser zur Geltung zu bringen, wurde dafür eine eigene UCopperList erstellt. Beim Wechseln vom ersten Screen in den zweiten verhindert die kleine Pause Schwierigkeiten zwischen dem Basic-Interpreter und der System-Software.

Sicherlich haben Sie schon einige Ideen, wie Sie den Copper für Ihre eigenen Programme einsetzen können. Es muß ja nicht gleich so bunt zugehen wie in unserem letzten Beispiel. Oft bringt schon ein geteilter Bildschirm den gewünschten Effekt. Sogar ein einfaches Grafikbild als »Hintergrundfarbe« ist denkbar. – Über eines haben Sie auf jeden Fall Gewißheit. Der von Ihnen programmierte Copper läuft unter Basic ebenso schnell wie in jeder anderen Programmier-Sprache.

Kapitel 19

Programme

In diesem Kapitel finden Sie einige Programme, die für die zugehörigen Teile des Buches etwas zu umfangreich ausgefallen sind. Da mit ihnen aber bestimmte Techniken und Verfahren recht eindrucksvoll demonstriert werden, will ich sie Ihnen nicht vorenthalten.

19.1 Motor

Das erste Programm unserer kleinen Sammlung befaßt sich mit der Farb-Animation. Es stellt die schematisierte Funktion eines Viertakt-Otto-Motors dar. Die Bewegung der Kolben, das Öffnen und Schließen der Ein- und Auslaßventile und die Zündung des Gas-Luft-Gemisches wird durch die Technik der Farb-Animation dargestellt. Die einzelnen Programmschritte mit den zugehörigen Variablen-Bezeichnungen sind einfach zu verfolgen. Es erübrigt sich daher eine zusätzliche Programmbeschreibung.

```
REM Motor  Pfad: Spezial/19Programme/Motor
'P19-1
SCREEN 1,320,256,5,1
WINDOW 2,,,0,1
DIM we%(255):GOSUB rau
FOR i=0 TO 3: WAVE i,we%:NEXT :ERASE we%
GOSUB Farben
GOSUB Konstruktion
COLOR 31
LOCATE 27,13:PRINT "Leertaste = ENDE"
LOCATE 28,13:PRINT "F1  =  schneller"
LOCATE 29,13:PRINT "F2  =  langsamer"
tim = .005:a=-1
WHILE a
  GOSUB Zuendung1
  Taste
```

```
 GOSUB Zuendung2
  Taste
  GOSUB Zuendung3
  Taste
  GOSUB Zuendung4
  Taste
WEND
WINDOW CLOSE 2
SCREEN CLOSE 1
END

Farben:
FOR i = Ø TO 31: READ fa%,r,g,b:PALETTE fa%,r,g,b:NEXT
r1=.4:g1=.6:b1=1    'blau
r2=1:g2=.6:b2=.67   'rot
r3=.33:g3=.87:b3=Ø  'gruen
w1=Ø                'schwarz
w2=.6               'grau
w3=1                'weiss
RETURN

Konstruktion:
'************
Ventile:
f=2
FOR i = 1Ø TO 25Ø STEP 8Ø
  FOR j = Ø TO 3Ø STEP 3Ø
    LINE (i+j,51)-(i+j+2Ø,54),f,bf
    LINE (i+j+9,4Ø)-(i+j+11,5Ø),f,bf
    f=f+1
NEXT j,i

ZylKopf:  LINE (Ø,2Ø)-(31Ø,4Ø),1,bf

KolbenAD:
FOR j = Ø TO 24Ø STEP 24Ø
  f=1Ø
  FOR i = 7Ø TO 15Ø STEP 1Ø
    LINE (j,i)-(j+7Ø,i+1Ø),f,bf
    f=f+1
NEXT i,j
```

```
KolbenBC:
FOR j = 8Ø TO 16Ø STEP 8Ø
  f=19:fa=23
  FOR i = 7Ø TO 1ØØ STEP 1Ø
    LINE (j,i)-(j+7Ø,i+1Ø),f,bf
    LINE (j,i+5Ø)-(j+7Ø,i+6Ø),fa,bf
    f=f+1:fa=fa+1
NEXT i,j

Kolbenbuchsen:
FOR i = 7Ø TO 23Ø STEP 8Ø
  LINE (i,4Ø)-(i+1Ø,17Ø),1,bf
  LINE (i+11,11Ø)-(i+8Ø,12Ø),14,bf
NEXT i

Blitz:
f=27
FOR i = 1 TO 241 STEP 8Ø
  LINE (i+22,41)-(i+47,49),f,bf
  LINE (i+31,5Ø)-(i+39,6Ø),f,bf
  LINE (i,55)-(i+68,69),f,bf
  f=f+1
NEXT i
RETURN

Animation:
'*********
Zuendung1:
PALETTE 27,w3,w3,w3:PALETTE 4,w1,w1,w1
PALETTE 7,r2,g2,b2:PALETTE 9,w1,w1,w1
PALETTE 8,r3,g3,b3:Pause
PALETTE 27,w1,w1,w1
GOTO Bewegung1

Zuendung2:
PALETTE 3,r2,g2,b2:PALETTE 28,w3,w3,w3
PALETTE 7,w1,w1,w1:PALETTE 6,r3,g3,b3
PALETTE 8,w1,w1,w1:Pause
PALETTE 28,w1,w1,w1
GOTO Bewegung2
```

```
Zuendung3:
PALETTE 3,w1,w1,w1:PALETTE 2,r3,g3,b3
PALETTE 5,r2,g2,b2:PALETTE 6,w1,w1,w1
PALETTE 30,w3,w3,w3:Pause
PALETTE 30,w1,w1,w1
GOTO Bewegung1

Zuendung4:
PALETTE 2,w1,w1,w1:PALETTE 5,w1,w1,w1
PALETTE 4,r3,g3,b3:PALETTE 29,w3,w3,w3
PALETTE 9,r2,g2,b2:Pause
PALETTE 29,w1,w1,w1
GOTO Bewegung2

Bewegung1:
p=22
FOR i = 10 TO 13
  PALETTE i,w1,w1,w1    'alte Pos loeschen
  PALETTE i+5,r1,g1,b1  'neue Pos setzen
  PALETTE p,r1,g1,b1    'alte Pos setzen
  PALETTE p+4,w1,w1,w1  'neue Pos loeschen
  p=p-1:Pause
NEXT i
RETURN

Bewegung2:
p=19
FOR i = 13 TO 10 STEP-1
  PALETTE i,r1,g1,b1
  PALETTE i+5,w1,w1,w1
  PALETTE p,w1,w1,w1
  PALETTE p+4,r1,g1,b1
  p=p+1:Pause
NEXT i
RETURN

rau:
FOR i=0 TO 255
  we%(i)=INT(RND*127)
NEXT i
RETURN
```

```
SUB Taste STATIC
  SHARED tim,a
  ta-=INKEY-
  IF ta-=CHR-(32) THEN a=Ø
  IF ta-=CHR-(129) THEN tim=tim-.ØØ5:IF tim <.ØØ5 THEN tim=.ØØ5
  IF ta-=CHR-(13Ø) THEN tim=tim+.ØØ5:IF tim >1 THEN tim=1
  dau=tim*8Ø:IF dau>1.5 THEN dau=1.5
  SOUND 25,dau,255,t
  t=t+1:IF t>3 THEN t=Ø
END SUB

SUB Pause STATIC
  SHARED tim
tt#=TIMER
WHILE TIMER<tt#+tim:WEND
END SUB

DATA Ø,Ø,Ø,Ø,      1,.6,.6,.6, 2,.53,.87,Ø, 3,1,.6,.67
DATA 4,.53,.87,Ø, 5,1,.6,.67, 6,.53,.87,Ø, 7,1,.6,.67
DATA 8,.53,.87,Ø, 9,1,.6,.67, 1Ø,.4,.6,1,  11,.4,.6,1
DATA 12,.4,.6,1,  13,.4,.6,1, 14,.4,.6,1,  15,Ø,Ø,Ø
DATA 16,Ø,Ø,Ø,    17,Ø,Ø,Ø,   18,Ø,Ø,Ø,    19,Ø,Ø,Ø
DATA 2Ø,Ø,Ø,Ø,    21,Ø,Ø,Ø,   22,Ø,Ø,Ø,    23,.4,.6,1
DATA 24,.4,.6,1,  25,.4,.6,1, 26,.4,.6,1,  27,Ø,Ø,Ø
DATA 28,Ø,Ø,Ø,    29,Ø,Ø,Ø,   3Ø,Ø,Ø,Ø,    31,1,1,.13
```

Gefällt Ihnen das Programm? Oder noch besser gefragt, hat es Sie dazu animiert, eigene Programme mit Farb-Animationen zu entwickeln? Prima! Dann hat es seinen Zweck erfüllt.

19.2 CAVE, Höllenfahrt durch die Höhle

Das folgende Programm zeigt, wie mit recht geringen Mitteln durch das PlayField-Scrolling ein, wie ich meine, recht gelungenes Spiel verwirklicht werden kann.

19.2.1 CAVE, das Programm

```
REM Cave  Pfad: Spezial/19Programme/cave
'P19-2
CLEAR
DEFINT a-z :tiefe=1
DIM yo(1Ø23),yu(1Ø23),we(255),txt-(4),bp&(tiefe)
```

```
sh=PEEKW(PEEKL(WINDOW(7)+46)+14)
sb=32Ø:tiefe=1:vm=Ø:tp=79:tp1=15:stit&=Ø
wb=sb:wh=sh:idf&=Ø:fl&=4Ø96:wt&=Ø
GOSUB Zeichen
GOSUB LibraryOeffnen
GOSUB Speicher      :IF fehl THEN ende
GOSUB Bitmapanlegen :IF fehl THEN ende
CALL Schirm (sb,sh,tiefe,vm,tp,stit&,mb&) :IF fehl THEN ende
CALL Fenster (wb,wh,idf&,fl&,wt&,tp1) :IF fehl THEN ende
CALL Zeigerdaten
GOSUB zeichnen

start:
POKEW ri&+8,Ø
CALL RethinkDisplay&
CALL ScrollVPort&(vp&)
GOSUB Requester
IF jn&=Ø THEN ende
a=-1
WHILE a
    x=PEEKW(win&+14):y=PEEKW(win&+12)
    IF y>yo(x) THEN a=Ø
WEND
CALL SetPointer&(win&,m2&,8,16,-15,Ø)

a=-1:Dx=Ø:sw=1:pu=Ø
WHILE a
    POKEW ri&+8,Dx
    CALL RethinkDisplay&
    CALL ScrollVPort&(vp&)
    x=PEEKW(win&+14):y=PEEKW(win&+12)
    Dx=Dx+sw
    z=z+sw:IF z>5Ø THEN z=Ø:SOUND 1ØØØ,1,255
    IF Dx >1Ø23 THEN Dx=Ø:pu=pu+1:sw=sw+1:IF pu>5 THEN a=Ø
    x1=x+Dx :IF x1>1Ø23 THEN x1=x1-1Ø23
    IF y<yo(x1) OR y+8>yu(x1) THEN a=Ø
WEND
txt-(1)="    Sie haben "+CHR-(pu+48)+" Punkte erreicht"+CHR-(Ø)
IF pu=6 THEN
  txt-(2)="Gratulation zur maximalen Punktzahl"+CHR-(Ø)
  FOR j=1 TO 1Ø
    FOR i= 5ØØ TO 98Ø STEP 3Ø:r=(i-5ØØ)/3Ø
```

```
      CALL SetRGB4&(vp&,1,r,8,7):SOUND i,.5,255,1
    NEXT
    FOR i= 980 TO 500 STEP-30:r=(i-500)/30
      CALL SetRGB4&(vp&,1,r,8,7):SOUND i,.5,255,1
    NEXT
  NEXT j
  CALL SetRGB4&(vp&,1,13,8,7)
ELSE
  txt-(2)="   Maximale Punktzahl = 6 Punkte"+CHR-(0)
  FOR i = 0 TO 15
    CALL SetRGB4&(vp&,17,i,i,15)
    CALL SetRGB4&(vp&,18,15,i,i)
    CALL SetRGB4&(vp&,19,i,15,i)
    SOUND 35-i,.5,255,1
  NEXT i
  CALL PointerFarben (0,0,0)
  FOR i=1 TO 5000:NEXT
  CALL PointerFarben (15,15,15)
END IF
CALL ClearPointer&(win&)
GOTO start

ende:
IF win& THEN CALL CloseWindow&(win&)
IF scr& THEN CALL CloseScreen&(scr&)
FOR i = 0 TO tiefe-1
  IF bp&(i) THEN CALL FreeRaster&(bp&(i),bmbr,bmRows)
NEXT
IF rk& THEN CALL FreeRemember&(rk&,-1)
IF fehl THEN
  ON fehl GOSUB f1,f2,F3
  BEEP:PRINT ft-
END IF
ERASE yo,yu,we,txt-,bp&
LIBRARY CLOSE
END

f1:ft-= "Fehler bei OpenWindow":RETURN
f2:ft-= "Speicher nicht ausreichend":RETURN
F3:ft-= "FEHLER bei OpenScreen":RETURN
```

```
LibraryOeffnen:
DECLARE FUNCTION AllocRaster&() LIBRARY
DECLARE FUNCTION AllocRemember&() LIBRARY
DECLARE FUNCTION OpenScreen&() LIBRARY
DECLARE FUNCTION OpenWindow&() LIBRARY
DECLARE FUNCTION AutoRequest&() LIBRARY
LIBRARY ":bue/intuition.library"
LIBRARY ":bue/graphics.library"
RETURN

Speicher:
art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
ms&=AllocRemember&(rk&,4Ø,art&)  'fuer NScreen-Struktur
IF ms&=Ø THEN fehl=2 :RETURN
mw&=AllocRemember&(rk&,48,art&)  'fuer NWindow-Struktur
IF mw&=Ø THEN fehl=2 :RETURN
mb&=AllocRemember&(rk&,1ØØ,art&) 'fuer Bitmap-Struktur
IF mb&=Ø THEN fehl=2 :RETURN
m1&=AllocRemember&(rk&,14Ø,art&) 'fuer Requester-Text
IF m1&=Ø THEN fehl=2 :RETURN
m2&=AllocRemember&(rk&,5Ø,art&)  'fuer Sprite-Daten
IF m2&=Ø THEN fehl=2
RETURN

zeichnen:
CALL SetRGB4&(vp&,Ø,Ø,Ø,Ø)
CALL SetRGB4&(vp&,1,13,8,7)
CALL PointerFarben (15,15,15)
CALL SetRast&(rp&,1)
CALL SetAPen&(rp&,Ø)
FOR j=Ø TO 5
  FOR i=Ø TO 12
    READ x,y,x2,y2
    CALL Move&(rp&,x+j,y+j)
    CALL Draw&(rp&,x2+j,y2+j)
  NEXT i
  RESTORE Titel
NEXT j
CALL SetAPen&(rp&,1)
tc-=" M&T by Horst-Rainer Henning ":cv&=SADD(tc-)
CALL Move&(rp&,52,17Ø):CALL Text& (rp&,cv&,29)
CALL SetAPen&(rp&,Ø)
```

```
dmin=65:go=1Ø:gu=sh-dmin:d=dmin
anfang=sh/2-2Ø:y1=anfang
FOR x=Ø TO 95Ø
  IF y1>gu THEN y1=gu
  IF y1<go THEN y1=go
  yo(x)=y1:yu(x)=y1+d
  y1=y1+RND*9-RND*9:d=RND*5+dmin
NEXT
diff!=anfang-y1:diff!=diff!/73
FOR x1=Ø TO 72  'restliche Strecke
  y2=y1+diff!*x1:y2=y2+RND*7-RND*7
  d=RND*5+dmin
  yo(x+x1)=y2:yu(x+x1)=y2+d
NEXT
CALL SetRast&(rp&,1)
FOR x=Ø TO 1Ø23
  CALL Move&(rp&,x,yo(x))
  CALL Draw&(rp&,x,yu(x))
NEXT
CALL SetAPen&(rp&,1)
FOR i=Ø TO 255:we(i)=RND*127:NEXT:WAVE 1,we
RETURN

Bitmapanlegen:
bmbr=1Ø24
bmBytPerRow=bmbr/8      'Bytes per Grafikzeile
bmRows=sh               'Grafik-Zeilen
volum&=bmBytPerRow*bmRows
FOR i = Ø TO tiefe-1
  bp&(i)=AllocRaster&(bmbr,bmRows)
  IF bp&(i)=Ø THEN fehl=2:RETURN
  CALL BltClear&(bp&(i),volum&,Ø)
NEXT
POKEW mb&,bmBytPerRow         'Bytes/Reihe
POKEW mb&+2,bmRows            'Reihen
POKE mb&+5,tiefe              'Tiefe
FOR n = Ø TO tiefe-1
  POKEL mb&+8+(n*4),bp&(n)    'n BitPlanes
NEXT
'CALL InitBitmap&(mb&,tiefe,bmbr,bmRows)
RETURN
```

```
Requester:
CALL Texte (SADD(txt-(Ø)),Ø,Ø,m1&+2Ø)
CALL Texte (SADD(txt-(1)),4Ø,2Ø,m1&+4Ø)
CALL Texte (SADD(txt-(2)),5Ø,4Ø,n&)
CALL Texte (SADD(txt-(3)),Ø,6Ø,n&)
CALL Texte (SADD(txt-(4)),Ø,8Ø,n&)
jn&= AutoRequest&(win&,m1&,m1&+6Ø,m1&+8Ø,Ø,Ø,31Ø,13Ø)
RETURN

SUB Texte (te&,y%,o%,n&) STATIC
SHARED m1&
POKE m1&+o%,1
POKE m1&+o%+2,2    :POKEW m1&+o%+4,5 :POKEW m1&+o%+6,5+y%
POKEL m1&+o%+12,te&         :POKEL m1&+o%+16,n&
END SUB

Zeichen:
txt-(Ø)="             C A V E"+CHR-(Ø)
txt-(1)="    bringen Sie das Raumschiff "+CHR-(Ø)
txt-(2)="    fehlerlos durch die Hoehle"+CHR-(Ø)
txt-(3)="START"+CHR-(Ø)
txt-(4)="ENDE"+CHR-(Ø)
RETURN

SUB Schirm (sb%,sh%,tie%,vm%,tp%,stit&,sbm&) STATIC
SHARED ms&,scr&,rp&,vp&,fehl%,ri&
StrukturNeuerScreen:
POKEW ms&,Ø      :POKEW ms&+2,Ø      'linke u.obere Kante
POKEW ms&+4,sb% :POKEW ms&+6,sh%    'Breite, Hoehe
POKEW ms&+8,tie%                    'Tiefe
POKE ms&+1Ø,Ø    :POKE ms&+11,1      'DetailPen, BlockPen
POKEW ms&+12,vm%                    'ViewModes
POKEW ms&+14,tp%                    'Type  CUSTOMSCREEN
CUSTOMBITMAP
POKEL ms&+16,Ø                      'TextAttr
POKEL ms&+2Ø,stit&                  'Screen-Titel
POKEL ms&+24,Ø  :POKEL ms&+28,sbm& 'Gadget,Bitmap
scr&= OpenScreen&(ms&)
IF scr&=Ø THEN fehl%=3
rp&=scr&+84 :vp&=scr&+44:ri&=PEEKL(scr&+8Ø)
END SUB
```

```
SUB Fenster (wb%,wh%,idf&,fl&,wtit&,tp%) STATIC
SHARED mw&,scr&,win&,fehl%
StrukturNeuesFenster:
POKEW mw&,Ø      :POKEW mw&+2,Ø      'linke u.obere Kante
POKEW mw&+4,wb% :POKEW mw&+6,wh%    'Breite, Hoehe
POKE mw&+8,1     :POKE mw&+9,Ø       'DetailPen, BlockPen
POKEL mw&+1Ø,idf&                    'IDCMPFlag
POKEL mw&+14,fl&                     'Flags ACTIVATE
POKEL mw&+26,wtit&                   'Window-Titel
POKEL mw&+3Ø,scr&                    'Screen
POKEW mw&+38,1ØØ:POKEW mw&+4Ø,3Ø    'min. Breite und Hoehe
POKEW mw&+42,wb%:POKEW mw&+44,wh%   'max. Breite und Hoehe
POKEW mw&+46,tp%                     'Type  CUSTOMSCREEN
win&= OpenWindow&(mw&)
IF win&=Ø THEN fehl%=1
END SUB

SUB Zeigerdaten STATIC
  SHARED m2&
  FOR i = Ø TO 42 STEP 2
    READ sd%:POKEW m2&+i,sd% AND 65535&
  NEXT i
END SUB

SUB PointerFarben (r%,g%,b%) STATIC
SHARED vp&
CALL SetRGB4&(vp&,17,Ø,Ø,b%) 'Zeigerfarbe 2 blau
CALL SetRGB4&(vp&,18,r%,Ø,Ø) 'Zeigerfarbe 3 rot
CALL SetRGB4&(vp&,19,Ø,g%,Ø) 'Zeigerfarbe 4 gruen
END SUB

Rakete:
DATA &hØ,&hØ
DATA &HØ,&H8ØØØ,&H7c,&HeØØ4,&H3fe,&Hf8Ø6
DATA &hfff,&hfØØ7,&Hfff,&HfØØ7,&H3fe,&Hf8Ø6
DATA &H7C,&HEØØ4,&HØ,&H8ØØØ
DATA &HØ,&HØ,&HØ,&HØ

Titel:
DATA 5Ø,5Ø,1ØØ,5Ø,5Ø,5Ø,5Ø,15Ø,5Ø,15Ø,1ØØ,15Ø
DATA 11Ø,5Ø,11Ø,15Ø,11Ø,5Ø,16Ø,5Ø,16Ø,5Ø,16Ø,15Ø
DATA 11Ø,1ØØ,16Ø,1ØØ,17Ø,5Ø,195,15Ø,22Ø,5Ø,195,15Ø
DATA 23Ø,5Ø,28Ø,5Ø,23Ø,1ØØ,28Ø,1ØØ,23Ø,15Ø,28Ø,15Ø
DATA 23Ø,5Ø,23Ø,15Ø
```

19.2.2 CAVE, die Programmbeschreibung

Wie gewohnt dimensionieren wir zuerst die Variablen-Felder und legen die Variablen für den Screen und das Fenster fest. Als Farbtiefe wählen wir 1. Das hat aber nichts mit Speicherproblemen zu tun. Auch zwei BitPlanes hätten spielend Platz. Das Programm kommt jedoch mit zwei Farben zurecht. Mehr wäre in diesem Falle zu viel. Nach dem Einlesen der Strings für den Requester öffnen wir erst einmal die Intuition- und Grafik-Library. In der Subroutine »Speicher« reservieren wir die Speicherbereiche für die Strukturen NewScreen, NewWindow, Bitmap und IntuitionText und für die Daten des neuen Maus-Sprites. Damit können wir bereits die Bitmap anlegen. Wir wählen eine Breite von 1024 Pixel und als Höhe die Screen-Höhe.

In dem Unterprogramm »Schirm« setzen wir für die Screen-Type die Flags für CUSTOMSCREEN UND CUSTOMBITMAP. Natürlich muß auch das Feld Bitmap mit der Adresse der Bitmap versorgt werden. Das Fenster erhält dieses Mal keine IDCMP-Flag zugewiesen. Warum das so ist, erkläre ich etwas später. Im Typen-Feld wird natürlich ebenfalls das Flag für CUSTOMSCREEN gesetzt. Nach dem Einlesen der neuen Zeigerdaten geht es weiter zur Subroutine »zeichnen«.

Hier setzen wir zuerst mit *SetRGB4* die Hintergrundfarbe auf Schwarz und die Vordergrundfarbe auf einen bräunlichen Farbton. In einem eigenen Unterprogramm erhält der Mauszeiger die Farben Rot, Grün und Blau. Nachdem der ganze Raster mit der Vordergrundfarbe gefüllt wurde (*SetRast*), wird die Zeichenfarbe mit *SetAPen* schwarz eingestellt. Nun kann in zwei geschachtelten Schleifen der Titel gezeichnet werden. Anschließend werden die obere und die untere Begrenzung des Tunnels für alle 1024 X-Positionen mittels Zufallszahlen ermittelt und in Variablenfeldern gespeichert. Die Berechnung der Zufallszahlen ist so angelegt, daß der Tunnel einen wirklichkeitsnahen, gezackten Rand erhält und gleichzeitig in der Y-Achse auf- und absteigt. Damit sich der Tunnel am Ende der Bitmap mit dem Tunnelanfang auch trifft, werden die letzten 73 X-Positionen langsam an den Anfangswert angeglichen:

```
diff!=anfang-y1:diff!=diff!/73
```

Mit den berechneten Werten kann nun die Höhle mit *Move* und *Draw* in den RastPort des Screens gezeichnet werden. Am Schluß der Routine »zeichnen« wird noch die Wellenform für ein weißes Rauschen festgelegt.

Beim Label »start« beginnt endlich das Hauptprogramm. Nachdem die Grafik in die Startposition (Dx=0) gebracht wurde, wird ein Requester ausgegeben. Der Anwender wird gefragt, ob er das Programm starten oder beenden will. Die Intuition-Routine *AutoRequest* gibt wahr oder unwahr zurück, je nachdem welche der beiden Tasten vom Anwender selektiert wurden. Wurde wahr, also *start* ausgewählt, so begibt sich das Programm in der WHILE/WEND-Schleife in Lauerstellung. Die Speicherstellen 14 und 12

verraten, wo sich der Mauszeiger gerade befindet. Sobald der Mauspfeil den Tunnel erreicht hat, wird mit *SetPointer* aus dem Mauszeiger eine kleine Rakete.

Gleichzeitig beginnt in der folgenden WHILE/WEND-Schleife der Scrollvorgang. Dazu wird *Dx* in die 8. Speicherstelle der RasInfo-Struktur gepoket. Mit *RethinkDisplay* und *ScrollViewPort* kommt der Screen auf die neue Position der Bitmap. Aus der Window-Struktur wird die aktuelle Mausposition geholt (*x* und *y*). *Dx* für den nächsten Durchgang und die Variable *sw* (Schwierigkeit) werden erhöht. Der Zeiger *z* zählt mit, wann die nächste SOUND-Anweisung ausgeführt werden soll. Wenn über die gesamte Bitmap gerollt wurde (*Dx*>1023), wird *Dx* wieder auf Null gesetzt, die Punktzahl *pu* und *sw* um eins erhöht. Nach 6 Durchgängen (*pu*>5) wird die Schleifenvariable *a* logisch unwahr. Am Ende der Schleife wird überprüft, ob eine Kollision zwischen Sprite und Höhlenwand stattgefunden hat und gegebenenfalls die Schleifen-Variable ebenfalls auf Null gesetzt. Dazu wird die Sprite-Position mit der oberen Höhlengrenze (IF y<yo(xl)) und die Sprite-Position+Sprite-Höhe mit der unteren Höhlenbegrenzung (OR y+8>yu(xl)) verglichen.

Nach dem Verlassen der Schleife wird zuerst der Screen wieder an die Nullposition der Bitmap gelegt. Je nach der erreichten Punktzahl explodiert entweder das Raumschiff, oder eine Sieges-Sirene mit flackerndem Hintergrund zeigt das Erreichen des Zieles an. Das Ergebnis wird in Strings festgehalten (für den Requester) und zu *start* zurückgesprungen.

19.2.3 CAVE, der Programmablauf

Es ist eine Schande für jeden Programmierer, wenn ein Spielprogramm zusätzlich eine schriftliche Beschreibung braucht. (Leider hat sich das noch nicht bei allen Programmierern herumgesprochen.) Dieses Unterkapitel soll deshalb nicht erklären, wie das Programm funktioniert, sondern warum was wie gemacht wurde.

Damit der Eindruck einer unendlichen langen Höhle entsteht, wurde als View-Modus LORES gewählt. Bei einer Bitmap-Breite von 1024 Pixel passen dadurch immerhin 3,2 Bildschirmbreiten nebeneinander. Das Zeichnen der Höhle wurde in zwei Abschnitte zerlegt. Zuerst wird das Titelbild gezeigt und dabei die Höhlengrenzen berechnet. Damit entfällt eine notwendige Warteschleife und der Anwender merkt nichts von dem Rechenvorgang. Anschließend kann das Titelbild gelöscht und die Zeichnung der Höhle, aus den Arrays heraus, sehr schnell erfolgen.

Mit dem Erscheinen des Requesters ist das Start-Bild für dieses und für jedes weitere Spiel fertig. Durch einen Mausklick auf das ENDE-Gadgets kann das Programm ordnungsgemäß verlassen werden. Ein Mausklick auf *START* beginnt das Programm, indem zuerst der Requester verschwindet. Da die Höhe des Requesters bzw. des START-Gadgets so gewählt wurde, daß der Mauspfeil in diesem Augenblick in der Höhle liegt,

beginnt sofort das Programm. Aus dem Mauspfeil wird eine kleine Rakete, und das Bild beginnt nach links zu rollen. Dabei ist in regelmäßigen Abständen das Tropfgeräusch eines auf den Höhlenboden fallenden Wassertropfens zu hören.

Vielleicht haben Sie sich schon Gedanken darüber gemacht, warum die Mausposition nicht über den IDCMP abgefragt wird. Die Programmierung wäre kein Problem, jedoch hat die Sache einen Haken. Da der Spieler während des Spielablaufes keine Maustaste drücken muß, könnte man in der NewWindow-Struktur nur durch das IDCMP-Flag MOUSEMOVE eine Meldung an den MessagePort auslösen.

Was passiert aber, wenn der Spieler seine Maus nicht bewegt? Die Rakete würde ohne jegliche Kollision friedlich mitten durch das Gebirge spazieren.

Die ganze Arbeit des Spielers besteht darin, die Rakete von den Höhlenwänden fernzuhalten. Das geht am Anfang recht gemächlich und ohne Schwierigkeiten vor sich. Hat sich der Spieler gerade an das Bummeltempo gewöhnt (nach einem Umlauf um die Bitmap), so verdoppelt sich plötzlich die Scroll-Geschwindigkeit. (Nun wird um 2 Pixel je Schleifendurchgang gescrollt). Die Anzahl der fallenden Wassertropfen verdoppelt sich natürlich zur akustischen Unterstreichung des Tempos auch. Logischerweise braucht auch der zweite Umlauf nur die halbe Zeit. Damit erhöht sich das Tempo erneut. Das geht so weiter, bis mit 6 Pixel pro Schleifendurchgang gescrollt wird. Da muß die Maus blitzschnell bewegt werden. Das schnell wiederkehrende Tropfgeräusch zerrt dabei auch an den Nerven.

Haben Sie es schon bis zum Siegesbild gebracht? Auf Anhieb sicherlich nicht. Wenn ja, haben Sie die Siegessirene mit den synchron laufenden Farbänderungen des Hintergrundes bereits bewundern dürfen. Wenn nicht, wissen Sie, wie ein Sprite mit einfachen Mitteln »explodieren« kann. Dabei werden in einer Schleife die drei Farbbestandteile Rot, Grün und Blau der drei Sprite-Farben jeweils um 1 in der *SetRGB4*-Routine erhöht. Dazwischen wird jedesmal eine SOUND-Anweisung aufgerufen, die einen Teil des Explosionsgeräusches repräsentiert. Zum Schluß ist der Sprite weiß. Die Farbe des Maus-Sprites ändert sich nun in Schwarz und ist auf dem schwarzen Untergrund nicht mehr zu sehen. Nachdem die Rakete sich aufgelöst hat, kommt wieder der normale Mauspfeil zum Vorschein. Den brauchen wir ja wieder für das Startbild.

Sollten Sie die Schwierigkeit des Spieles steigern wollen, so bringt die Erhöhung der Scroll-Geschwindigkeit kein gutes Bild mehr. Besser bietet sich eine Reduzierung des Höhlenquerschnittes an. Dazu brauchen Sie nur die Variable *dmin* in der Subroutine »zeichnen« etwas zu reduzieren. Ich hoffe, daß das kleine Spiel Ihnen viel Freude bereitet und Ihnen eine Anregung für eigene Schöpfungen gegeben worden ist.

19.3 TopIcon, der Icon-Editor

Wie im Kapitel über die Icons versprochen, finden Sie anschließend einen Icon-Editor. Da er in Basic geschrieben ist, können Sie ihn nach Belieben verändern und Ihren Wünschen und Erfordernissen anpassen. Die Sonder-Icons auf der Programm-Diskette, die diesem Buch beigefügt ist, wurden alle auf diesem Icon-Editor erstellt. Sie zeigen Ihnen dadurch einige Möglichkeiten für eigene Schöpfungen auf.

19.3.1 TopIcon, das Programm

```
REM TopIcon  Pfad: Spezial/19Programme/TopIcon
'P19-3
SYSTEM
'by Horst-Rainer Henning
IF FRE(-1)<14ØØØØ& THEN PRINT "Speicher zu klein":END
PRINT "BITTE warten"
CLEAR ,75ØØØ&
ON ERROR GOTO korrektur
DEFINT a-z :f=Ø
DIM text-(14)
GOSUB LibraryOeffnen
CALL StringGadgetInitialisieren :IF f THEN ende
GOSUB InitTemp :IF f THEN ende
Xp=5ØØ:Yd=4Ø:Yp1=1Ø:Yp2=5Ø:Yp3=9Ø:Yp4=13Ø
CLS:PRINT "TOP ICON"
PRINT :PRINT "Anweisungen ueber Menues eingeben"
GOSUB Beschriftung
text-="":wbr=35Ø:wh=1ØØ
WINDOW 2,"T O P   I C O N",(Ø,5Ø)-(wbr,8Ø+wh),17
Fenster&=WindowLimits&(WINDOW(7),Ø,Ø,wbr+25,145)
GOSUB MenuAnlegen
ON MENU GOSUB MenuKontr :MENU ON
ON MOUSE GOSUB MausKontr:MOUSE ON
CALL Grafikzeiger    '1.Aufruf einschalten

a=-1
WHILE a
  SLEEP
  IF f THEN a=Ø
WEND
```

```
CALL Grafikzeiger    '2.Aufruf ausschalten
WINDOW CLOSE 2

ende:
  IF bPlane& THEN CALL FreeRaster&(bPlane&,640,200)
  IF rk& THEN CALL FreeRemember&(rk&,-1)
  IF rk2& THEN CALL FreeRemember&(rk2&,-1)
  LIBRARY CLOSE
  IF f THEN
    BEEP : PRINT "Speicher nicht ausreichend"
    tim&=TIMER:WHILE TIMER<tim&+5:WEND
  END IF
SYSTEM

ende2:PRINT " = Fehler-Nummer ":BEEP:GOTO ende

korrektur:
  IF ERR <>53 THEN
    PRINT ERR;:RESUME ende2
  ELSE
    BEEP:neu=-1:RESUME BildHolen
  END IF

LibraryOeffnen:
  DECLARE FUNCTION AllocRemember&() LIBRARY
  DECLARE FUNCTION AllocRaster&() LIBRARY
  DECLARE FUNCTION AddGadget&() LIBRARY
  DECLARE FUNCTION RemoveGadget&() LIBRARY
  DECLARE FUNCTION Request&() LIBRARY
  DECLARE FUNCTION GetMsg&() LIBRARY
  DECLARE FUNCTION TextLength&() LIBRARY
  DECLARE FUNCTION AutoRequest&() LIBRARY
  DECLARE FUNCTION WindowLimits&() LIBRARY
  LIBRARY ":bue/graphics.library"
  LIBRARY ":bue/exec.library"
  LIBRARY ":bue/intuition.library"
RETURN

MenuAnlegen:
  MENU 1,0,1,"Color"
  MENU 1,1,1,"  1 = blau   ":MENU 1,2,2,"  2 = weiss  "
  MENU 1,3,1,"  3 = schwarz":farbe=1
  MENU 1,4,1,"  4 = rot    "
```

```
  MENU 2,Ø,1,"Copy"
  MENU 2,1,2,"  Muster 2 >< Bild 1" :m2=1
  MENU 2,2,1,"  Muster 2 >< Bild 2"
  MENU 2,3,1,"  Muster 1 >< Bild 1"
  MENU 2,4,1,"  Muster 1 >< Bild 2"
  MENU 2,5,1,"  verschieben  -->  "
  MENU 2,6,1,"  verschieben  <--  "
  MENU 2,7,1,"  verschieben   ^   "
  MENU 2,8,1,"  verschieben   \/  "
  MENU 3,Ø,1,"Text":schr=1
  MENU 3,1,2,"  Topaz 8"
  MENU 4,Ø,1,"Disc"
  MENU 4,1,2,"  Muster laden":m4=1
  MENU 4,2,1,"  Icon laden  "
  MENU 4,3,1,"  speichern ":MENU 4,4,1,"  E N D E   "
  MENU 5,Ø,1,"Tool":MENU 5,1,2,"  Punkt  " :m5=1
  MENU 5,2,1,"  Linie  ":MENU 5,3,1,"  fuellen"
  MENU 5,4,1,"  neu    ":MENU 5,5,1,"  Text   "
  MENU 6,Ø,1,"Draw"
  MENU 6,1,2,"  Bild 1" :Bild=1: MENU 6,2,1,"  Bild 2"
  MENU 7,Ø,1,"Size":MENU 7,1,2,"  * 1" :fakt=1
  MENU 7,2,1,"  * 2":MENU 7,3,1,"  * 3"
  MENU 7,4,1,"  * 4":MENU 7,5,1,"  * 5"
  MENU 8,Ø,1,"Select"
  MENU 8,1,2,"  Doppelbild" :m6=1
  MENU 8,2,1,"  invers    "
  MENU 9,Ø,1,"Typ":MENU 9,1,1,"  1=Diskette   "
  MENU 9,2,1,"  2=Verzeichnis"
  MENU 9,3,1,"  3=Programm   "
  MENU 9,4,1,"  4=Datei      "
  MENU 9,5,1,"  5=Muelleimer "
RETURN

MausKontr:
  GOSUB position
  startX=posx:startY=posy
  ON m5 GOSUB Punkt,Linie,fuellen,neu,Texte
RETURN
```

```
MenuKontr:
  MenuTitel=MENU(Ø)
  MenuNummer=MENU(1)
  ON MenuTitel GOTO Farben,Kopie,Schrift,Disk,Tool,Draw,size,
**                                                   Select,Type

Farben:
  FOR i =1 TO 4:MENU 1,i,1:NEXT
  farbe=MenuNummer-1:MENU 1,MenuNummer,2
RETURN

Schrift:
  MENU 3,schr,2
RETURN

Disk:
  MENU 4,m4,1 :m4=MenuNummer
  ON m4 GOSUB BildLesen,AltIcon,speichern,beenden
  MENU 4,m4,2
RETURN

Tool: MENU 5,m5,1 :m5=MenuNummer:MENU 5,m5,2:RETURN

size:MENU 7,fakt,1 :fakt=MenuNummer:MENU 7,fakt,2:RETURN

Select:MENU 8,m6,1:m6=MenuNummer:MENU 8,m6,2:RETURN

Type: RETURN

Kopie:
MENU 2,m2,1 :m2=MenuNummer:MENU 2,m2,2
Xvon=Xp:Xzu=Xp
WINDOW OUTPUT 1:RaFa=Ø:GOSUB Rahmen
IF m2 <3 THEN Yvon=Yp4 ELSE Yvon=Yp3
IF m2 =1 OR m2=3 THEN Yzu=Yp1 ELSE Yzu=Yp2
IF m2>4 THEN
  IF Bild=1 THEN Yzu =Yp1 :Yvon=Yp1 ELSE Yzu=Yp2: Yvon=Yp2
  IF m2>6 THEN
    IF m2=8 THEN
      hoehe=hoehe+1
      Yzu=Yzu+1
    ELSE
      Yzu=Yzu-1
    END IF
  ELSE
```

```
    IF m2=6 THEN
      Xzu=Xzu-1
    ELSE
      breite=breite+1
      Xzu=Xzu+1
    END IF
  END IF
END IF
IcSp=INT((breite+15)/16)        'Breite in Worten
aa=3+((IcSp)*(hoehe+1)*2)
DIM Icon1(aa) :DIM Icon2(aa)
GET (Xvon,Yvon)-(Xvon+breite-1,Yvon+hoehe-1),Icon1
IF m2>4 THEN LINE(Xvon,Yvon)-(Xvon+breite,Yvon+hoehe),Ø,bf
IF m2<5 THEN GET (Xzu,Yzu)-(Xzu+breite-1,Yzu+hoehe-1),Icon2
PUT (Xzu,Yzu),Icon1,PSET
IF m2<5 THEN PUT (Xvon,Yvon),Icon2,PSET
ERASE Icon1:ERASE Icon2
RaFa=1:GOSUB Rahmen
WINDOW OUTPUT 2
RETURN

Rahmen:
  FOR i = Yp1+1Ø TO Yp4+1Ø STEP Yd
    LINE (Xp-2,i-12)-(Xp+breite+7,i-8+hoehe),RaFa,b
    LINE (Xp-1,i-11)-(Xp+breite+6,i-9+hoehe),RaFa,b
  NEXT
RETURN

Beschriftung:
  CALL SetAPen&(WINDOW(8),1)
  CALL SetBPen&(WINDOW(8),Ø)
  text-(Ø)="Bild 1:  Bild 2:  Muster 1:Muster 2:"
  tr&=SADD(text-(Ø)):tpos=Ø
  FOR i = Yp1+1Ø TO Yp4+1Ø STEP Yd
    CALL move&(WINDOW(8),Xp-8Ø,i)
    CALL text&(WINDOW(8),tr&+tpos,9)
    tpos=tpos+9
  NEXT
RETURN
```

```
Draw:
MENU 6,Bild,1:Bild=MenuNummer:MENU 6,Bild,2
Draw1:
REM pruefen ob Vergroessserung moeglich
WINDOW OUTPUT 2
IF breite*fakt >= WINDOW(2) THEN
    BEEP :zeig=1
    MENU 7,fakt,1 :fakt=fakt-1:MENU 7,fakt,2
  ELSE
    zeig=Ø
END IF
WINDOW OUTPUT 1
IF zeig THEN Draw1
IF Bild=1 THEN y1=Yp1 ELSE y1=Yp2
x1=Xp
FOR y2=y1 TO y1+hoehe
    x2=x1
    frag:
    x4=x2
    WINDOW OUTPUT 1
    farb= POINT(x2,y2)
    add:x2=x2+1 :IF x2>(x1+breite) THEN GOSUB trans :GOTO naechst
    IF POINT(x2,y2)=farb THEN add ELSE GOSUB trans :GOTO frag
    naechst:
NEXT y2
WINDOW OUTPUT 2
RETURN
trans:
  WINDOW OUTPUT 2
  x3=(x4-x1)*fakt:y3=(y2-y1)*fakt:x5=(x2-x1)*fakt
  LINE (x3,y3)-(x5+fakt-1,y3+fakt-1),farb,bf
RETURN

Punkt:
ok=1
WHILE MOUSE(Ø)<>Ø
  GOSUB position
  WINDOW OUTPUT 1
  xTest=posx/fakt:IF xTest>breite THEN GOSUB testX
  yTest=posy/fakt:IF yTest>hoehe THEN GOSUB testY
  x2=xTest+x1:y2=yTest+y1
```

```
  IF ok THEN PSET(x2,y2),farbe
    WINDOW OUTPUT 2
    x3=(x2-x1)*fakt:y3=(y2-y1)*fakt
  IF ok THEN LINE (x3,y3)-(x3+fakt-1,y3+fakt-1),farbe,bf
WEND
RETURN

testX:
IF xTest>13Ø THEN ok=Ø:RETURN
RaFa=Ø:GOSUB Rahmen
breite=xTest
RaFa=1:GOSUB Rahmen
RETURN

testY:
IF yTest>4Ø THEN ok=Ø:RETURN
RaFa=Ø:GOSUB Rahmen
hoehe=yTest
RaFa=1:GOSUB Rahmen
RETURN

Linie:
ok=1
WHILE MOUSE(Ø)<>Ø
  GOSUB position
  CALL SetDrMd&(WINDOW(8),3)
  LINE (startX,startY)-(posx,posy),farbe
  LINE (startX,startY)-(posx,posy),farbe
WEND
CALL SetDrMd&(WINDOW(8),1)
REM pset oder im rechten Bild zeichnen
xTest=posx/fakt:IF xTest>breite THEN testX
yTest=posy/fakt:IF yTest>hoehe THEN testY
WINDOW OUTPUT 1
IF ok THEN
  LINE ((startX/fakt)+x1,(startY/fakt)+y1)
**                 -(xTest+x1,yTest+y1),farbe
  GOSUB Draw1
ELSE
  BEEP
END IF
RETURN
```

```
fuellen:
  IF MOUSE(Ø)<>Ø THEN
    WINDOW OUTPUT 2
    GOSUB position
    POKEL (WINDOW(8)+12),mbm&
    CALL SetAPen&(WINDOW(8),farbe)
    CALL flood&(WINDOW(8),1,posx,posy)
    WINDOW OUTPUT 1
    x2=(posx/fakt)+x1:y2=(posy/fakt)+y1
    POKEL (WINDOW(8)+12),mbm&
    CALL SetAPen&(WINDOW(8),farbe)
    CALL flood&(WINDOW(8),1,x2,y2)
    WINDOW OUTPUT 2
  END IF
RETURN

neu:
  WINDOW OUTPUT 2:CLS
  IF Bild=1 THEN y1=Yp1 ELSE y1=Yp2
  x1=Xp:WINDOW OUTPUT 1
  LINE (x1,y1)-(x1+breite,y1+hoehe),Ø,bf
  WINDOW OUTPUT 2
RETURN

Texte:
ok=1:WINDOW OUTPUT 2
IF MOUSE(Ø)<>Ø THEN
  GOSUB position
  bpen= POINT(posx,posy)
  a–="":n=Ø :WINDOW OUTPUT 1
  CALL SetAPen&(WINDOW(8),farbe)
  CALL SetBPen&(WINDOW(8),bpen)
  taste:ta–=INKEY–:IF ta–="" THEN taste
    IF ta–<>CHR–(13) THEN a–=a–+ta–:n=n+1:GOTO taste
  tt&=SADD(a–)
  lg=TextLength&(WINDOW(8),tt&,n)
  xTest=posx/fakt+lg:IF xTest>breite THEN GOSUB testX
  yTest=posy/fakt:IF yTest>hoehe THEN GOSUB testY
  x2=posx/fakt+x1:y2=yTest+y1
  IF ok THEN
    WINDOW OUTPUT 1
    RaFa=Ø:GOSUB Rahmen
```

```
    IF x2+lg-x1>breite THEN breite=x2+lg-x1
    IF y2-y1>hoehe THEN hoehe=y2-y1
    CALL move&(WINDOW(8),x2,y2)
    CALL text&(WINDOW(8),tt&,n)
    RaFa=1:GOSUB Rahmen
    GOSUB Draw1
  ELSE
    BEEP
  END IF
END IF
CALL SetBPen&(WINDOW(8),0)
RETURN

position:
muss=MOUSE(0):posx=MOUSE(1):posy=MOUSE(2):RETURN

speichern:
  IF lesen THEN BEEP ELSE speichern1
  text-(1)="Sie haben kein Icon eingelesen."
  text-(2)="Hinweis: Vergleichsbilder muessen"
  text-(3)="vor dem Icon eingelesen werden!"
  text-(4)="weiter"
  text-(5)="IRRTUM"
  CALL Texte (text-(1),5,0,100,m1&+120)
  CALL Texte (text-(2),5,10,120,m1&+140)
  CALL Texte (text-(3),5,20,140,n&)
  CALL Texte (text-(4),5,0,160,n&)
  CALL Texte (text-(5),5,0,180,n&)
  jn&= AutoRequest&(WINDOW(7),m1&+100,m1&+160,m1&+180,0,0,300,70)
RETURN

speichern1:
  text-(6)="Verkleinern Sie das Fenster"
  text-(7)="auf die benoetigte Groesse."
  text-(8)="Groesse gilt fuer beide Icons!"
  text-(9)="fertig"
  text-(10)="IRRTUM"
  CALL Texte (text-(6),5,0,200,m1&+220)
  CALL Texte (text-(7),5,10,220,m1&+240)
  CALL Texte (text-(8),5,20,240,n&)
  CALL Texte (text-(9),5,0,260,n&)
  CALL Texte (text-(10),5,0,280,n&)
```

```
jn&= AutoRequest&(WINDOW(7),m1&+2ØØ,m1&+26Ø,m1&+28Ø,Ø,Ø,3ØØ,7Ø)
IF jn& THEN w=1 ELSE RETURN
WiBr=WINDOW(2)  :WiBr=WiBr/fakt
WiHo=WINDOW(3)  :WiHo=WiHo/fakt
IcSp=INT((WiBr+15)/16)       'Breite in Worten
WiBr=IcSp*16                 'Aufgerundet auf Wortlaenge
REM Zusammensetzen der info.Datei
REM DiskObjekt Laenge 78 Byte
  obj-=""
  FOR i =Ø TO 1Ø STEP 2
    obj-=obj-+MKI-(PEEKW(inp&+i))
  NEXT
  obj-=obj-+MKI-(WiBr)   'Breite
  obj-=obj-+MKI-(WiHo)   'Hoehe in Zeilen
  IF m6=1 THEN
    obj-=obj-+MKI-(6)     'Flag 2-Bild-Icon
  ELSE
    obj-=obj-+MKI-(flags)
  END IF
  obj-=obj-+MKI-(activation)
  obj-=obj-+MKI-(Type)
  obj-=obj-+MKL-(GadgetRender&)
  IF m6=1 THEN
    obj-=obj-+MKL-(1)            'Image Bild 2
  ELSE
    obj-=obj-+MKL-(Ø)            'kein Doppelbild
  END IF
  FOR i = 3Ø TO 76 STEP 2
    obj-=obj-+MKI-(PEEKW(inp&+i))
  NEXT
REM Drawer Laenge 56 Byte
  dra-=""
  IF IconTyp<3 OR IconTyp>4 THEN
    FOR i =78 TO 132 STEP 2
      dra-=dra-+MKI-(PEEKW(inp&+i))
    NEXT
  END IF
REM Image
  ima-(Ø)="" :ima-(1)=""
  IF m6=1 THEN n=1 ELSE n=Ø
  FOR i =Ø TO n
```

```
    ima-(i)=MKI-(Ø)                 'linke Kante
    ima-(i)=ima-(i)+MKI-(Ø)         'obere Kante
    ima-(i)=ima-(i)+MKI-(WiBr)      'Breite
    ima-(i)=ima-(i)+MKI-(WiHo)      'Hoehe
    ima-(i)=ima-(i)+MKI-(2)         'Tiefe
    ima-(i)=ima-(i)+MKL-(59664&)    'Zeiger auf Bitpl.Pseudowert
    ima-(i)=ima-(i)+CHR-(3)         'PlanePick
    ima-(i)=ima-(i)+CHR-(Ø)         'PlaneOnOff
    ima-(i)=ima-(i)+MKL-(Ø)'Zeig.naechst.ImageStr.nicht gesetzt
  NEXT i
REM Bitplanes erstes Icon
REM Feldgroesse in Worten
  WINDOW OUTPUT 1
  c=(WiBr/16)*WiHo*2
  aa=3+((WiBr/16)*(WiHo+1)*2)
  pl1-=""
  DIM Icon1(aa)
  wex=Xp+WiBr-1:wey=y11+WiHo-1
  GET (Xp,y11)-(wex,wey),Icon1
  FOR i = 3 TO c+2
    pl1-=pl1-+MKI-(Icon1(i))
  NEXT
  ERASE Icon1
REM Bitplanes zweites Icon
  pl2-=""
  IF m6=1 THEN
    DIM Icon2(aa)
    GET (Xp,y12)-(Xp+WiBr-1,y12+WiHo-1),Icon2
    FOR i = 3 TO c+2
      pl2-=pl2-+MKI-(Icon2(i))
    NEXT
    ERASE Icon2
  END IF
gesamt-=obj-+dra-+ima-(Ø)+pl1-+ima-(1)+pl2-
'DefaultTool + ToolTypes aus altem Icon uebernehmen
Rest-="":RestLaenge=Ø
InclEbenenAlt=stre+ben
'IF default& THEN
  FOR i =InclEbenenAlt TO Altlaenge-2 STEP 2
    Rest-=Rest-+MKI-(PEEKW(inp&+i))
  NEXT
```

```
  'END IF
  gesamt-=gesamt-+Rest-
  OPEN text- FOR OUTPUT AS #1
    PRINT#1,gesamt-;
  CLOSE #1
  kText-=text-+".info":KILL kText-
  WINDOW OUTPUT 2
NeuBeginn:
  IF rk2& THEN CALL FreeRemember&(rk2&,-1)
  WINDOW OUTPUT 1:CLS
  GOSUB Beschriftung
  WINDOW OUTPUT 2:CLS
  deltax=wbr-WINDOW(2)-5
  deltay=2ØØ-WINDOW(3)-7Ø
  CALL SizeWindow&(WINDOW(7),deltax,deltay)
RETURN

beenden:a=Ø:RETURN

BildLesen:
  y11=Yp3:y12=Yp4
  lesen=1
  GOSUB BildHolen
RETURN

AltIcon:
  lesen=Ø
  y11=Yp1:y12=Yp2
BildHolen:
  IF neu THEN
    CLOSE #1:neu=Ø
    WINDOW 2,"T O P   I C O N",(Ø,5Ø)-(wbr,85+wh),1
  END IF
  WINDOW OUTPUT 1
  CALL StringGadgetInitialisieren2
  CALL RequesterOn (text-)
  IF pl&<>64 THEN RETURN
  tele=LEN(text-)
  IF RIGHT-(text-,6)=".info " THEN text-=LEFT-(text-,tele-1)
  IF RIGHT-(text-,5)<>".info" THEN text-=text-+".info"
  RaFa=Ø:GOSUB Rahmen
```

```
AltesObjektEinlesen:
  'Struktur DiskObject  Laenge 78 Byte
  OPEN text- FOR INPUT AS 1
    magic=CVI(INPUT-(2,1))
    Version=CVI(INPUT-(2,1))
    nextGadget&=CVL(INPUT-(4,1))
    linkeKante=CVI(INPUT-(2,1))
    obereKante=CVI(INPUT-(2,1))
    breite=CVI(INPUT-(2,1))
    hoehe=CVI(INPUT-(2,1))
    flags=CVI(INPUT-(2,1))
    activation=CVI(INPUT-(2,1))
    Type=CVI(INPUT-(2,1))
    GadgetRender&=CVL(INPUT-(4,1))
    SelectRender&=CVL(INPUT-(4,1))
    dummy-=INPUT-(18,1)
    IconTyp=ASC(INPUT-(1,1))
    dummy-=INPUT-(1,1)
    default&=CVL(INPUT-(4,1))
    Tool&=CVL(INPUT-(4,1))
    Xpos&=CVL(INPUT-(4,1))
    Ypos&=CVL(INPUT-(4,1))
    Drawer&=CVL(INPUT-(4,1))
    ToolWind&=CVL(INPUT-(4,1))
    Stack&=CVL(INPUT-(4,1))
  CLOSE #1
'Typ des Icons ins Menu schreiben
  FOR i = 1 TO 5:MENU 9,i,1:NEXT
  MENU 9,IconTyp,2
'Flag ins Menu schreiben
  MENU 8,1,1:MENU 8,2,1
  IF flags=6 THEN m6=1 ELSE m6=2
  MENU 8,m6,2
'berechnen der Speichergroesse des input-Puffers
  ben=INT((breite+15)/16)       'Breite in Worten
  ben=ben*hoehe*4           '2 Bit-Ebenen eines Icons in Bytes
  IF SelectRender& THEN ben1=ben*2 ELSE ben1=ben
  ben1=ben1+5ØØ
  rek2&=Ø:rk2&=VARPTR(rek2&)
  inp&=AllocRemember&(rk2&,ben1,Art&)
  IF inp&=Ø THEN f=-1:RETURN
```

```
  offs=Ø
  OPEN text- FOR INPUT AS 1
    Altlaenge=LOF(1)
    WHILE NOT EOF(1)
      POKEW inp&+offs,CVI(INPUT-(2,1))
      offs=offs+2
    WEND
  CLOSE #1
  IF Drawer& THEN st=56+78 ELSE st=78 'Offset bis image
  str=st+2Ø        'Offset bis 1.Bitebene
  POKEL inp&+st+1Ø,inp&+str
  IF GadgetRender& THEN CALL DrawImage&(WINDOW(8),inp&+st,Xp,y11)
  IF SelectRender& THEN
    stre=str+ben+2Ø   'Offset bis 2.Bitebene
    POKEL inp&+st+1Ø+ben+2Ø,inp&+stre
    CALL DrawImage&(WINDOW(8),inp&+st+ben+2Ø,Xp,y12)
  ELSE
    stre=str
  END IF
  RaFa=1:GOSUB Rahmen
  WINDOW OUTPUT 2
IF lesen THEN
  IF rk2& THEN CALL FreeRemember&(rk2&,-1)
END IF
RETURN

SUB StringGadgetInitialisieren STATIC
  SHARED m1&,w&,rk&,Art&,f%
  w&=WINDOW(7)
  Art&=3+(2^16):rek&=Ø:rk&=VARPTR(rek&)
  CALL initMemory       :IF f% THEN EXIT SUB
  CALL initStringGadget
  CALL initBoolGadget
  CALL initBool
  CALL initBool
  CALL initRequester
END SUB

SUB StringGadgetInitialisieren2 STATIC
  SHARED m1&,w&,rk&,Art&,f%
  w&=WINDOW(7)
  CALL initStringGadget
```

```
  CALL initBoolGadget
  CALL initBool
  CALL initBool
  CALL initRequester
END SUB

SUB initMemory STATIC
  SHARED rk&,Art&,f%,mØ&,m1&,m2&,m3&,m4&,m5&,m7&,mg&,mb&,mz&,mr&
  mØ&=AllocRemember&(rk&,1ØØ,Art&)
  IF mØ&=Ø THEN f%=-1:EXIT SUB        'Buffer
  m2&=AllocRemember&(rk&,1ØØ,Art&)
  IF m2&=Ø THEN f%=-1:EXIT SUB        'UndoBuffer
  m3&=AllocRemember&(rk&,2Ø,Art&)
  IF m3&=Ø THEN f%=-1:EXIT SUB        'Render
  m5&=AllocRemember&(rk&,3Ø,Art&)
  IF m5&=Ø THEN f%=-1:EXIT SUB        'Border
  m4&=AllocRemember&(rk&,4Ø,Art&)
  IF m4&=Ø THEN f%=-1:EXIT SUB        'StringInfo
  mg&=AllocRemember&(rk&,15Ø,Art&)
  IF mg&=Ø THEN f%=-1:EXIT SUB        'Gadget
  m1&=AllocRemember&(rk&,3ØØ,Art&)
  IF m1&=Ø THEN f%=-1:EXIT SUB        'IText
  mr&=AllocRemember&(rk&,12Ø,Art&)
  IF mr&=Ø THEN f%=-1:EXIT SUB        'Requester
  m7&=AllocRemember&(rk&,62,Art&)
  IF m7&=Ø THEN f%=-1:EXIT SUB        'LinienFeld
  mb&=AllocRemember&(rk&,3Ø,Art&)
  IF mb&=Ø THEN f%=-1:EXIT SUB        'border
  mz&=AllocRemember&(rk&,1ØØ,Art&)
  IF mz&=Ø THEN f%=-1                 'fuer SpriteDaten
END SUB

SUB initStringGadget STATIC
  SHARED mØ&,mg&,m1&,m2&,m3&,m4&,m5&,f%
  SrukturRender:
    POKEL m3&,Ø     :POKEW m3&+4,2Ø1: POKEW m3&+6,Ø
    POKEW m3&+8,2Ø1:POKEW m3&+1Ø,1Ø: POKEL m3&+12,1Ø
    POKEL m3&+16,Ø

  StrukturBorder:
    POKEW m5&,-1     : POKEW m5&+2,-1      'LeftEdge/TopEdge
    POKE m5&+4,1     : POKE m5&+5,3        'FrontPen/BackPen
```

```
    POKE m5&+6,1     : POKE m5&+7,5       'DrawMode/Count
    POKEL m5&+8,m3& : POKEL m5&+12,Ø     'XY/NextBorder
  StrukturStringInfo:
    POKEL m4&+Ø,mØ& : POKEL m4&+4,m2&    'Buffer/UndoBuffer
    POKEW m4&+8,1    : POKEW m4&+1Ø,99    'BufferPos/MaxChars
    POKEW m4&+12,1   'DispPos
  StrukturGadget:
    POKEL mg&,mg&+44 : POKEW mg&+4,4Ø     'NextG/LeftEdge
    POKEW mg&+6,3Ø   : POKEW mg&+8,2ØØ    'TopEdge/Width
    POKEW mg&+1Ø,1Ø  : POKEW mg&+12,Ø     'Height/Flags
    POKEW mg&+14,517 : POKEW mg&+16,41ØØ '/GadgetType
    POKEL mg&+18,m5& : POKEL mg&+34,m4&  'Render/SpecialI
END SUB

SUB initRequester STATIC
  SHARED mr&,mg&,m1&,text-()
  text-(11)="Bitte geben Sie den Pfadnamen"
  text-(12)="des gewuenschten Icons ein."
  nx&=Ø
  a%=Ø:b%=Ø:c%=1Ø:CALL Texte (text-(11),c%,a%,b%,m1&+2Ø)
  a%=1Ø:b%=2Ø:c%=2Ø:CALL Texte (text-(12),c%,a%,b%,nx&)
  StrukturRequester:
    POKEL mr&,Ø      :POKEW mr&+4,36Ø  'OlderRequest/linke
    POKEW mr&+6,Ø    :POKEW mr&+8,27Ø  'obere Kante/Breite
    POKEW mr&+1Ø,1ØØ:POKEW mr&+12,Ø    'Hoehe/RelLeft
    POKEW mr&+14,Ø   :POKEL mr&+16,mg& 'RelTop/ReqGadget
    POKEL mr&+2Ø,Ø   :POKEL mr&+24,m1& 'ReqBorder/ReqText
    POKEW mr&+28,Ø   :POKE mr&+3Ø,3    'Flags/BackFill
END SUB

SUB Texte (txte-,x%,y%,o%,n&) STATIC
  SHARED m1&
  ttt-=txte-+CHR-(Ø)
  StrukturIText:
    POKE m1&+o%,1           : POKE m1&+o%+1,Ø     'Front/BackPen
    POKE m1&+o%+2,2         : POKEW m1&+o%+4,x% 'DrawMode/Left
    POKEW m1&+o%+6,5+y%      : POKEL m1&+o%+8,Ø 'Top/TextAttr
    POKEL m1&+o%+12,SADD(ttt-): POKEL m1&+o%+16,n&'text/Next
END SUB

SUB initBoolGadget  STATIC
  SHARED mb&,m7&
  LinienFeld:
```

```
  RESTORE
  FOR i = Ø TO 29
    READ w%:POKEW m7&+(2*i),w%
  NEXT
  DATA 4Ø,5,6Ø,5,9Ø,2Ø,9Ø,3Ø,6Ø,45,4Ø,45,1Ø,3Ø
  DATA 1Ø,2Ø,4Ø,5,32,1Ø,2Ø,1Ø,2Ø,4Ø,8Ø,4Ø,8Ø,1Ø,2Ø,1Ø
  StrukturBorder:
    POKEW mb&,Ø       : POKEW mb&+2,Ø  'LeftEdge/TopEdge
    POKE mb&+4,1      : POKE mb&+5,3   'FrontPen/BackPen
    POKE mb&+6,1      : POKE mb&+7,15  'DrawMode/Count
    POKEL mb&+8,m7& : POKEL mb&+12,Ø 'XY/NextBorder
END SUB

SUB initBool  STATIC
  SHARED m1&,w&,rk&,Art&,mg&,mb&,text-()
  IF di%=Ø THEN
    di%=15Ø :x%=27:o%=6Ø:ng&=Ø
    ofs%=88:text-(13)="IRRTUM":ak%=6'262
    txte-=text-(13)+CHR-(Ø)
  ELSE
    di%=Ø :x%=42:o%=4Ø:ng&=mg&+88
    ofs%=44:text-(14)="OK":ak%=1
    txte-=text-(14)+CHR-(Ø)
  END IF

  StrukturIText1:
    POKE m1&+o%,1         :POKE m1&+o%+1,Ø       'FrontPen/BackPen
    POKE m1&+o%+2,2       :POKEW m1&+o%+4,x%     'DrawMode/LeftEdge
    POKEW m1&+o%+6,22     :POKEL m1&+o%+8,Ø      'TopEdge/TextAttr
    POKEL m1&+o%+12,SADD(txte-):POKEL m1&+o%+16,Ø'text/NextText
  StrukturGadget:
    POKEL mg&+ofs%,ng&     : POKEW mg&+ofs%+4,1Ø+di%'NextG/linke
    POKEW mg&+ofs%+6,45    : POKEW mg&+ofs%+8,1ØØ     'obere/Breite
    POKEW mg&+ofs%+1Ø,5Ø   : POKEW mg&+ofs%+12,2      'Hoehe/Flags

    POKEW mg&+ofs%+14,ak%: POKEW mg&+ofs%+16,4Ø97 'Activat/Gad
    POKEL mg&+ofs%+18,mb&: POKEL mg&+ofs%+22,Ø     'Render/Select
    POKEL mg&+ofs%+26,m1&+o%    'GadgetText
END SUB

SUB RequesterOn (txt-) STATIC
  SHARED mØ&,mr&,w&,m4&,pl&
  lg=LEN(txt-)
```

```
  FOR i=1 TO lg
    t-=MID-(txt-,i,1):POKE mØ&+(i-1),ASC(t-)
  NEXT
  POKE mØ&+i,Ø          'abgeschlossen mit NULL
  rq&= Request&(mr&,w&) :me&=PEEKL(w&+86)
  abfrage:
    mess&=GetMsg&(me&):IF mess&=Ø THEN abfrage
    pl&=PEEKL(mess&+2Ø):CALL ReplyMsg&(mess&)
  IF pl&<>64 THEN zurueck
  lng=PEEKW(m4&+16) :txt-=""
    FOR i=Ø TO lng-1:txt-=txt-+CHR-(PEEK(mØ&+i)) :NEXT
  zurueck:
  IF PEEKW(mr&+28)>8191 THEN CALL EndRequest&(mr&,w&)
END SUB

SUB Grafikzeiger STATIC
  SHARED mz&
  IF zeiger THEN aus
    FOR i = Ø TO 71 STEP 2
      READ sd%:POKEW mz&+i,sd% AND 65535&
    NEXT i
  zeigen:
    DATA &hØ,&hØ,&H44Ø,&HC6Ø,&H44Ø,&HC6Ø,&H44Ø,&HC6Ø
    DATA &hØ44Ø,&hØc6Ø,&H44Ø,&HFC7E,&HFC7E,&HFC7E
    DATA &hØ,&hØ,&H1ØØ,&HØ,&HØ,&HØ,&HFC7E,&HFC7E
    DATA &hØ44Ø,&hfc7e,&H44Ø,&HC6Ø,&H44Ø,&HC6Ø
    DATA &hØ44Ø,&hØc6Ø,&H44Ø,&HC6Ø,&HØ,&HØ,&HØ,&HØ
    CALL SetPointer&(WINDOW(7),mz&,15,15,-8,-7)
    zeiger=-1:EXIT SUB
  aus:
    CALL ClearPointer&(WINDOW(7))
    RESTORE zeigen
END SUB

InitTemp:
b=64Ø:h=2ØØ
bmBytPerRow=b/8 'Bytes per Grafikzeile
volum&=bmBytPerRow*h
  bPlane&=AllocRaster&(b,h)
  IF bPlane&=Ø THEN f=-1:RETURN
  CALL BltClear&(bPlane&,volum&,Ø)
'Fuer Struktur TmpRas
```

```
mbm&=AllocRemember&(rk&,2Ø,Art&)
POKEL mbm&,bPlane&
POKEL mbm&+4,volum&
CALL InitTmpRas&(mbm&,bPlane&,volum&)
rp1&=PEEKL(WINDOW(7)+5Ø)
POKEL rp1&+12,mbm&
'CALL RethinkDisplay&
RETURN
```

19.3.2 Arbeiten mit TopIcon

Beginnen will ich mit einem Hinweis zu Ihrer Sicherheit bzw. der Ihrer Disketten. Legen Sie von der Diskette, auf der Sie ein oder mehrere Icons ändern wollen, eine Kopie an und arbeiten Sie nur mit der Kopie. Wenn Sie nach dem Ändern feststellen, daß alles in Ordnung ist, können Sie die Kopie als Original einsetzen. Aus diesem Grund wurde dem Programm eine kleine Sperre verpaßt, die ein ungewolltes Aktivieren verhindern soll. Laden Sie deshalb das Programm und setzen Sie die dritte Programmzeile

```
SYSTEM
```

außer Kraft. Nun können Sie das Programm starten. Das Programm benötigt einige Sekunden, bevor es alle Subroutinen und Unterprogramme durchlaufen hat. Warten Sie, bis aus dem Mauspfeil ein grafischer Cursor geworden ist. Zum Testen der einzelnen Optionen laden Sie am besten ein normales Programm-Icon. Wählen Sie dazu aus dem Menü *Disc* die Option *Icon laden* aus. Nach einem kurzen Augenblick erscheint auf der rechten Bildhälfte ein Requester, der die Eingabe des Icon-Namens erwartet. Klicken Sie das Feld des String-Gadgets an und tippen Sie den Namen des Icons ein. Den Zusatz *.info* brauchen Sie nicht mit eingeben, da das vom Programm erledigt wird.

Wenn Sie den Namen fertiggeschrieben haben, drücken Sie entweder RETURN oder klicken Sie in das OK-Feld des Requesters. Der Requester verschwindet und das Icon wird in das Feld bei *Bild 1:* gezeichnet. Zur Überprüfung des Icon-Typs wählen Sie das Menü *Typ*. Sie sehen, daß als Typ *4=Datei* mit einem Haken versehen ist. In diesem Menü können Sie keine Änderung vornehmen. Links daneben ist das Menü *Select*, bei dem *invers* einen Haken hat. Es handelt sich also um ein Icon, daß bei seiner Selektierung invers dargestellt wird. Dieser Menü-Punkt kann geändert werden. Wir kommen etwas später darauf zurück. Wieder links daneben finden Sie das Menü *size*. Mit ihm können Sie festlegen, mit welcher Vergrößerung ins Arbeitsfenster gezeichnet werden soll. Sollten Sie eine Vergrößerung wählen, die für das Fenster zu groß ist, korrigiert das Programm den Wert. Das Fenster auf der linken Seite mit dem Namen TOPICON ist das Arbeitsfenster.

Nachdem Sie eine Vergrößerung festgelegt haben, rufen Sie vom Menü *Draw* die Option *Bild 1* auf. Das Bild 1 wird nun in der gewünschten Vergrößerung ins Arbeitsbild gezeichnet. Damit sind wir schon bei dem Menü *Tool*, mit den einzelnen Zeichenwerkzeugen. Bei allen Optionen können Sie im Menü *color* zwischen den 4 Standardfarben wählen. Mit *Punkt* setzen Sie, durch Drücken der linken Maustaste, einen oder mehrere Punkte. Die Zeichnung erfolgt in der richtigen Vergrößerung im Arbeitsfeld und gleichzeitig im Bild 1, dem Original. Wollen Sie Ihr Zeichenfeld vergrößern, welches Sie am Rahmen um Bild 1 erkennen, so setzen Sie einfach einen Punkt in der Hintergrundfarbe rechts neben oder unter die Zeichnung im Arbeitsfenster. Der Rahmen vergrößert sich, innerhalb der erlaubten Grenzen, sofort. Das zweite Zeichenwerkzeug ist *Linie*. Damit wird eine Linie zwischen dem Punkt gezeichnet, wo Sie die Maustaste gedrückt haben, und dem Punkt, wo Sie diese wieder losgelassen haben. Nach der Ausführung müssen Sie einen Augenblick warten, da das Bild neu gezeichnet wird.

Zum Füllen von Flächen rufen Sie *fuellen* auf. Daß dabei die Umrandung der zu füllenden Fläche ohne Belang ist, versteht sich inzwischen von selbst. Gefällt Ihnen Ihr Kunstwerk nicht besonders, so erhalten Sie mit *neu* wieder eine freie Zeichenfläche. Vom Original des Icons ist dann allerdings auch nichts mehr zu sehen. Mit dem letzten *Tool* können Sie einen Text auf der Grafik des Piktogrammes ausgeben. Dazu markieren Sie mit dem Mauspfeil bzw. dem grafischen Cursor die untere linke Ecke, ab und über der der Text beginnen soll. Achten Sie dabei auf die Farbe des Bildpunktes an dieser Position, denn er wird zur Wahl der Hintergrundfarbe des Textes herangezogen. Die Schriftfarbe wählen Sie wie gewohnt aus dem Menü *Color*. Der Text wird erst ausgegeben, wenn die Eingabe durch RETURN abgeschlossen ist.

Damit können Sie Ihr erstes Icon abspeichern. Wählen Sie dazu *speichern* aus dem Menü *disc*. Es erscheint ein Requester, der Sie auffordert, das Arbeitsfenster zu verkleinern. Damit legen Sie die Abmessungen des Icons fest. Achten Sie dabei auf die rechte Fensterbegrenzung, die wegen des Größen-Gadgets etwas Abstand zur Außenkante des Icons benötigt. Wenn Sie fertig sind, klicken Sie das *fertig*-Gadget des Requesters an, und das geänderte Icon wird abgespeichert. Zum Schluß wird alles für den nächsten Durchgang vorbereitet. Wenn Sie nun das Programm mit *ENDE* aus dem Menü *Disc* verlassen, so finden Sie noch die alte Grafik des Icons vor. Erst wenn Sie das Fenster schließen und wieder öffnen, können Sie Ihr Werk bewundern.

Ein Doppelbild-Icon konstruieren

Nun werden wir aus dem normalen Programm-Icon aus unserem ersten Versuch ein Doppelbild-Icon zaubern. Lesen Sie dazu das Icon wie besprochen ein. Nun rufen Sie aus dem Menü *Draw* die Option *Bild 2* auf. Da sich im Feld für *Bild 2* zur Zeit keine Grafik befindet, dient diese Auswahl nur der Vorbereitung zum Zeichnen. Nun zeich-

nen Sie das zweite Bild für das Piktogramm in das Arbeits-Fenster. Achten Sie dabei darauf, daß die Größenverhältnisse mit Bild 1 übereinstimmen. Wenn Sie fertig sind, wählen Sie aus dem Menü *Select* den Menü-Punkt *Doppelbild* aus. Nun können Sie die Grafik wieder wie besprochen mit dem Menü-Punkt *speichern* aus dem *Disc*-Menü abspeichern.

Grafiken anderer Icons als Zeichenvorlage

Natürlich können Sie zur Konstruktion Ihrer Grafiken auch bestehende Grafiken einbeziehen. Dabei ist nur wichtig, daß Sie die richtige Reihenfolge beachten. Zuerst laden Sie die Hilfsgrafik mit dem Menü-Punkt *Muster laden*. Das Icon wird in das Feld Muster 1 und bei einem Doppelbild-Icon zusätzlich in das Feld Muster 2 gezeichnet. Jetzt können Sie mit *Icon laden* das Icon einlesen, das Sie ändern wollen. Löschen Sie komplett den alten Namen in dem String-Gadget und geben Sie den richtigen Namen ein.

Nun können Sie mit dem Menü *Copy* die einzelnen Bilder beliebig vertauschen. Außerdem können Sie auch die Lage der Grafik durch vier Menü-Punkte um je einen Bildpunkt in jede Richtung verschieben.

Ändern eines Disketten-Icons

Der Weg zu einem neuen Disketten-Icon ist auch nicht komplizierter. Auch hier und speziell hier sollten Sie die Änderung nur an einer Kopie der Diskette vornehmen.

Am Anfang gehen Sie wie gewohnt vor. Am einfachsten ist es, wenn sich TopIcon auf der Diskette befindet, dessen Icon ein neues Gesicht bekommen soll. Legen Sie die Diskette ins Laufwerk und starten TopIcon. Als Pfadnamen für das Disketten-Icon tippen Sie ein:

```
disk.info
```

Nun können Sie Ihr neues Icon auf die gewohnte Art und Weise zeichnen und abspeichern. Nach der Rückkehr auf die Workbench schließen Sie die Fenster der geänderten Diskette, nehmen die Diskette aus dem Diskettenschacht und warten, bis das Icon erlischt. Wenn Sie nun die Diskette neu einlegen, trägt das Icon die neue Zeichnung.

Grafik mit AMIGA-BASIC

Anhang

5. Teil

Commodore Sachbuch

Anhang I

Strukturen

I.1 Screen-Strukturen

Struktur NewScreen			
scr&= OpenScreen&(Speicher&)			
Länge	**Bezeichnung**	**Adresse**	
Wort	LeftEdge	scr&	linke Ecke des Screens (Null)
Wort	TopEdge	+2	obere Ecke des Screens in Pixel
Wort	Width	+4	Screen-Breite in Bildpunkten
Wort	Height	+6	Screen-Höhe max. 256/200
Wort	Depth	+8	Tiefe in BitPlanes 1–6
Byte	DetailPen	+10	Farbe für Titel-Text, Gadgets
Byte	BlockPen	+11	Farbe für Titel-Leiste (Hintergr.)
Wort	ViewModes	+12	View-Modus s.u.
Wort	Type	+14	Type
			CUSTOMSCREEN $F, CUSTOMBITMAP $40
Zeiger	Font	+16	Z. auf TextAttr-Struktur, oder 0
Zeiger	DefaultTitel	+20	Zeiger auf Titel-Text, oder 0
Zeiger	Gadgets	+24	Null
Zeiger	CustomBitmap	+28	Zeiger auf eigene Bitmap, oder 0

View-Modi

LORES (0) – Bildschirmbreite 320 Pixel
HIRES $8000 (32768) – Bildschirmbreite 640 Pixel
INTERLACE (4) – halbierte Bildfrequenz 512/400 Zeilen
SPRITES $4000 (16384) – Sprites
DUALPL $400 (1024) – zwei PlayFields
HAM $800 (2048) – Farbdarstellung durch HoldAndModify
PFBA $40 (64) – Priorität der PlayFields für DUALPF
HALFBRITE $80 (128)

Struktur Screen		
scr&=PEEKL(WINDOW(7)+46)		
Länge	**Bezeichnung**	**Adresse**
Zeiger	NextScreen	scr& Zeiger auf den nächsten Screen
Zeiger	FirstWindow	+4 Zeiger auf erstes Fenster des Screens
Wort	LeftEdge	+8 linke Ecke des Screens
Wort	TopEdge	+10 obere Ecke des Screens
Wort	Width	+12 Breite des Screens
Wort	Height	+14 Höhe des Screens
Wort	MouseY	+16 Maus-Position-Y zu oben links
Wort	MouseX	+18 Maus-Position-X zu oben links
Wort	Flags	+20 Flags siehe unten
Zeiger	Title	+22 Zeiger auf Titeltext
Zeiger	DefaultTitel	+26 Zeiger auf Fenstertext
Byte	BarHeight	+30 Höhe der Titelleiste
Byte	BarVBorder	+31 Titelleiste, vertikale Umrandung
Byte	BarHBorder	+32 Titelleiste, horizontale Umrandung
Byte	MenuVBorder	+33 Menüleiste, vertikale Umrandung
Byte	MenuHBorder	+34 Menüleiste, horizontale Umrandung
Byte	WBorTop	+35 Fenster, Oberkante der Umrandung
Byte	WBorLeft	+36 Fenster, linke Seite der Umrandung
Byte	WBorRight	+37 Fenster, rechte Kante der Umrandung
Byte	WBorBottom	+38 Fenster, Unterkante der Umrandung
Zeiger	TextAtribute	+40 Zeiger auf Font für Screen
Struktur ViewPort		
Zeiger	Next ViewPort	+44 Zeiger auf nächsten ViewPort
Zeiger	ColorMap	+48 Zeiger auf ColorMap-Struktur
.....		
Wort	Modes	+76 ViewPort-Modus
Zeiger	RasInfo	+80 Zeiger auf RasInfo-Struktur
Struktur RastPort		
Zeiger	Layer	+84 Zeiger auf Layer-Struktur
Zeiger	Bitmap	+88 Zeiger auf Bitmap-Struktur
.....		
Wort	TxSpacing	+148
Zeiger	RP__User	+150 Zeiger auf User-Rastport
Reserve	30 Byte	

Länge	Bezeichnung	Adresse
Struktur Bitmap		
Wort	BytesPerRow	+184 Anzahl der Bytes pro Zeile
Wort	Rows	+186 Anzahl der Grafik-Zeilen
.....		
Zeiger auf BitPlane		
Struktur LayerInfo		
Zeiger	top__layer	+224
Zeiger	check__lp	
.....		
Zeiger	LayerInfo__extra	
Zeiger	FirstGadget	+326 Z. auf 1. Gadget des Screens
Byte	DetailPen	+330 Farbe für Text d. Titelleiste
Byte	BlockPen	+331 Hintergrundfarbe, Titelleiste
Wort	SaveColor0	+332 Farbe für DisplayBeep
Zeiger	BarLayerList	+334 Z. auf Layer-Titelleiste
Zeiger	ExtData	+338 Zeiger für Anwender
Zeiger	UserData	+342 Zeiger für Anwender

Beachten Sie, daß es sich bei den Feldern für ViewPort, RastPort, Bitmap und LayerInfo nicht um Zeiger handelt. Dort sind deren komplette Strukturen abgelegt.

FLAGS (SCREENTYPE)

WBENCHSCREEN $01 – Flag für den Workbench-Bildschirm
CUSTUMSCREEN $0F (15) – Flag für Anwender-Bildschirm
SHOWTITLE $10 (16) – Flag für die Titelleiste
BEEPING $20 (32) – Intuition-Flag, wenn der Schirm BEEPt
CUSTOMBITMAP $40 (64) – Flag für Anwender-Bitmap.

I.2 Window-Strukturen

Struktur NewWindow			
win&= OpenWindow&(Speicher&)			
Länge	**Bezeichnung**	**Adresse**	
Wort	LeftEdge	win&	linke Kante des Fensters
Wort	TopEdge	+2	obere Kante des Fensters
Wort	Width	+4	Breite des Fensters
Wort	Height	+6	Höhe des Fensters
Byte	DetailPen	+8	Farbe für Titel-Text
Byte	BlockPen	+9	Farbe für Titel-Leiste
Langwort	IDCMPFlags	+10	IDCMP-Flags
Langwort	Flags	+14	Flags s.u.
Zeiger	FirstGadget	+18	Zeiger auf das erste Gadget
Zeiger	CheckMark	+22	Z auf Image für Markierungshaken
Zeiger	Title	+26	Zeiger auf Text für Fenster-Titel
Zeiger	Screen	+30	Zeiger auf Screen
Zeiger	Bitmap	+34	Zeiger auf Bitmap (Refresh)
Wort	MinWidth	+38	min. Breite des Fensters
Wort	MinHeight	+40	min. Höhe des Fensters
Wort	MaxWidth	+42	max. Breite des Fensters
Wort	MaxHeight	+44	max. Höhe des Fensters
Wort	Type	+46	Type s.u.

Flags (Window)

WINDOWSIZING $1	Größen-Gadget mit folgenden Flags: SIZEBRIGHT $10 (16) an rechter Fenstergrenze (voreingestellt) SIZEBOTTOM $20 (32) Größen-Gadget an unterer Fenstergrenze
WINDOWDEPTH $4	Flag für das Tiefen-Gadget
WINDOWCLOSE $8	Flag für ein Schließ-Gadget
WINDOWDRAG $2	Flag für Verschiebe-Gadget
GIMMEZEROZERO $400 (1024)	Nullpunkt innerhalb der Fenstergrenzen

Flags zum Auffrischen (wiederherstellen) von Fenstern:

SIMPLE__REFRESH $40 (64)	Nur Meldung an IDCMP
SMART__REFRESH $0	überlappte Fensterteile in Zwischenspeicher
SUPER__BITMAP $80 (128)	kompl. Fenster wird in Bitmap gespeichert
NOCAREREFRESH $20000	(131072) keine Auffrischung des Fensters
BACKDROP $100 (256)	für Hintergrund-Fenster
REPORTMOUSE $200 (512)	Mausbewegungen werden registriert
BORDERLESS $800 (2048)	für Fenster ohne Fensterbegrenzungen
ACTIVATE $1000 (4096)	Fenster wird aktives Fenster
RMBTRAP $10000 (65536)	registriert Drücken der rechten Maustaste

Typ

CUSTOMSCREEN ($F=15)	für Anwender-Screen
WBENCHSCREEN (1)	für Fenster auf Workbench

Struktur Window			
win&=WINDOW(7)			
Länge	**Bezeichnung**	**Adresse**	
Zeiger	NextWindow	win&	Zeiger auf das nächste Fenster
Wort	LeftEdge	+4	linke Kante des Fensters
Wort	TopEdge	+6	obere Kante des Fensters
Wort	Width	+8	Breite des Fensters
Wort	Height	+10	Höhe des Fensters
Wort	MouseY	+12	Maus-Position-Y zu oben links
Wort	MouseX	+14	Maus-Position-X zu oben links
Wort	MinWidth	+16	minimale Breite des Fensters
Wort	MinHeight	+18	minimale Höhe des Fensters
Wort	MaxWidth	+20	maximale Breite des Fensters
Wort	MaxHeight	+22	maximale Höhe des Fensters
Langwort	Flags	+24	siehe NewWindow-Struktur
Zeiger	MenuStrip	+28	Zeiger auf Menü
Zeiger	Title	+32	Zeiger auf Titel-Text
Zeiger	FirstRequest	+36	Zeiger auf den ersten Requester
Zeiger	DMRequest	+40	Zeiger auf Doppel-Klick Requester
Wort	ReqCount	+44	Requester-Zähler
Zeiger	WScreen	+46	Zeiger auf Screen dieses Fensters
Zeiger	RPort	+50	Zeiger auf den RastPort
Byte	BorderLeft	+54	Fenstergrenze links

Struktur Window			
win&=WINDOW(7)			
Länge	**Bezeichnung**	**Adresse**	
Byte	BorderTop	+55	Fenstergrenze oben
Byte	BorderRight	+56	Fenstergrenze rechts
Byte	BorderBottom	+57	Fenstergrenze unten
Zeiger	BorderRPort	+58	RastPort der Fenstergrenzen
Zeiger	FirstGadget	+62	Zeiger auf das erste Gadget
Zeiger	Parent	+66	
Zeiger	Descedant	+70	
Zeiger	Pointer	+74	Z.auf Sprite-Daten f. User-Pointer
Byte	PtrHeight	+78	Sprite-Höhe
Byte	PtrWidth	+79	Sprite-Breite (max.16)
Byte	XOffset	+80	Sprite X-Offset
Byte	YOffset	+81	Sprite Y-Offset
Langwort	IDCMPFlags	+82	IDCMP-Flags
Zeiger	UserPort	+86	Anwender-Port
Zeiger	WindowPort	+90	Intuition-Port
Zeiger	MessageKey	+94	
Byte	DetailPen	+98	Farbe für Titeltexte etc.
Byte	BlockPen	+99	Farbe für Titelleiste etc.
Zeiger	CheckMark	+100	Zeiger auf Image für User-Haken
Zeiger	ScreenTitle	+104	Zeiger auf Screen-Titel
Wort	GZZMouseX	+108	Mausposition-X bei GZZ
Wort	GZZMouseY	+110	Mausposition-Y bei GZZ
Wort	GZZWidth	+112	Fensterbreite bei GIMMEZEROZERO
Wort	GZZHeight	+114	Fensterhöhe bei GZZ
Zeiger	ExtData	+116	Zeiger auf Anwender-Daten
Zeiger	UserData	+120	Zeiger auf Anwender-Daten
Zeiger	WLayer	+124	Zeiger auf WLayer

I.3 Sonstige Strukturen der Video-Ausgabe

Struktur View,0		
v&=ViewAdress&		
Länge	**Bezeichnung**	**Adresse**
Zeiger	v__ViewPort	v& Z. auf den Viewport
Zeiger	v__LOFCprList	+4 Z. auf die Copper-Liste
Zeiger	v__SHFCprList	+8 Z. auf die Copper-Liste
Wort	v__DyOffset	+12 Y-Abstand zur oberen Ecke
Wort	v__DxOffset	+14 X-Abstand zur oberen Ecke
Wort	v__Modes	+16 View-Modi (siehe NewScreen-Struktur)

Struktur ViewPort,0		
vp&=ViewPortAdress&(WINDOW(7))		
Länge	**Bezeichnung**	**Adresse**
Zeiger	vp__Next	vp& Z. auf nächsten View-Port
Zeiger	vp__ColorMap	+4 Z. auf die Colormap
Zeiger	vp__DspIns	+8
Zeiger	vp__SprIns	+12
Zeiger	vp__ClrIns	+16
Zeiger	vp__UCopIns	+20 UserCopperList
Wort	vp__DWidth	+24 Breite
Wort	vp__DHeight	+26 Höhe
Wort	vp__DxOffset	+28
Wort	vp__DyOffset	+30
Wort	vp__Modes	+32 Modus (siehe NewScreen-Struktur)
Wort	vp__reserved	+34 Reserve
Zeiger	vp__RasInfo	+36 Zeiger auf RasInfo-Struktur

Struktur ColorMap,0		
cm&=PEEKL(vp&+4)		
Länge	**Bezeichnung**	**Adresse**
Byte	cm__Flags	cm&
Byte	cm__Type	+1
Wort	cm__Count	+2
Zeiger	cm__ColorTable	+4

Struktur ColorTable		
cp&=peekl(cm&+4)		
Länge	**Bezeichnung**	**Adresse**
Wort	Hintergrundfarbe	cp&
Wort	Farbe 1	+2
Wort	Farbe 2	+4
etc.		

Struktur Layer		
lay&=Screen&+224		
Länge	**Bezeichnung**	**Adresse**
Zeiger	Front	lay& Zeiger auf Vordergrund-Layer
Zeiger	Back	+4 Zeiger auf Hintergrund-Layer
Zeiger	ClipRect	+8 Zeiger auf ClipRect
Zeiger	RastPort	+12 Zeiger auf Layer-RastPort
Wort	MinX	+16 obere linke Ecke X
Wort	MinY	+18 obere linke Ecke Y
Wort	MaxX	+20 untere rechte Ecke X
Wort	MaxY	+22 untere rechte Ecke Y
Byte	Lock	+24
Byte	LockCount	+25
Byte	LayerLockCount	+26
Byte	reserved	+27 reserviert
Wort	reserved1	+28 reserviert
Wort	Flags	+30 Flags
Zeiger	SuperBitmap	+32 Zeiger auf Layer-SuperBitmap

Struktur Layer		
lay&=Screen&+224		
Länge	**Bezeichnung**	**Adresse**
Zeiger	SuperClipRect	+36
Zeiger	Window	+40 Window des Layers
Wort	Scroll__X	+44 Scroll-Position X in SuperBitmap
Wort	Scroll__Y	+46 Scroll-Position Y in SuperBitmap
Struktur	LockPort	+48
Struktur	LockMessage	+82
Struktur	ReplyPort	+102
Struktur	l__LockMessage	+136
Zeiger	DamageList	+156
Zeiger	cliprects	+160
Zeiger	LayerInfo	+164 Zeiger auf LayerInfo-Struktur
Zeiger	LayerLocker	+168
Zeiger	SuperSaverClipRects	+172
Zeiger	cr	+176
Zeiger	cr2	+180
Zeiger	crnew	+184
Zeiger	pl	+188

Struktur BitMap,0		
bm&=PEEKL(Window(8)+4)		
Länge	**Bezeichnung**	**Adresse**
Wort	bm__BytesPerRow	bm& Anzahl der Bytes pro Grafikzeile
Wort	bm__Rows	+2 Anzahl der Grafikzeilen
Byte	bm__Flags	+4 Bitmap-Flags
Byte	bm__Depth	+5 Tiefe, Anzahl der Ebenen
Wort	bm__Pad	+6 Pfad
Struktur	bm__Planes,8*4	+8 Z. auf die einzelnen BitPlanes

Struktur RasInfo,0		
ri&=PEEKL(vp&+36)		
Länge	**Bezeichnung**	**Adresse**
Zeiger	ri__Next	ri& Z. auf nächste RasInfo-Struktur
Zeiger	ri__BitMap	+4 Z. auf Bitmap
Wort	ri__RxOffset	+8
Wort	ri__RyOffset	+10

Struktur RastPort,0		
rp&=Window(8)		
Länge	**Bezeichnung**	**Adresse**
Zeiger	rp__Layer	rp& Z. auf Layer-Struktur
Zeiger	rp__BitMap	+4 Z. auf Bitmap-Struktur
Zeiger	rp__AreaPtrn	+8 Z. auf Areaptrn-Struktur
Zeiger	rp__TmpRas	+12 Z. auf Tmpras-Struktur
Zeiger	rp__AreaInfo	+16 Z. auf Areainfo-Struktur
Zeiger	rp__GelsInfo	+20 Z. auf Gelsinfo-Struktur
Byte	rp__Mask	+24 Maske
Byte	rp__FgPen	+25 Vordergrundfarbe
Byte	rp__BgPen	+26 Hintergrundfarbe
Byte	rp__AOLPen	+27 Randfarbe
Byte	rp__DrawMode	+28 Zeichen-Modus
Byte	rp__AreaPtSz	+29 Größe des Feldmusters
Byte	rp__Dummy	+30 leer
Byte	rp__linpatcnt	+31 Zähler für Linienmuster
Wort	rp__Flags	+32 Flags, siehe unten
Wort	rp__LinePtrn	+34 Linienmuster
Wort	rp__cp__xt	+36 Grafik-Cursor X-Position
Wort	rp__cp__yt	+38 Grafik-Cursor Y-Position
Struktur	rp__minterms,8	+40
Wort	rp__PenWidth	+48
Wort	rp__PenHeight	+50
Zeiger	rp__Font	+52 Z. auf Schriftart
Byte	rp__AlgoStyle	+56
Byte	rp__TxFlags	+57 Text-Flags
Wort	rp__TxHeight	+58 Texthöhe

Struktur RastPort,0		
rp&=Window(8)		
Länge	**Bezeichnung**	**Adresse**
Wort	rp__TxWidth	+60 Textbreite
Wort	rp__TxBaseline	+62 Textgrundlinie
Wort	rp__TxSpacing	+64
Zeiger	rp__RP__User	+66 Z. auf User-RastPort

Flags

`FRST_DOT,Ø (-1)`	
`ONE_DOT,1 (-2)`	Dieses Bit wird für den Einpunkt-Modus gesetzt.
`DBUFFER,2 (-4)`	Bit 2 steht für einen Double-Buffered-RastPort.
`AREAOUTLINE,3 (-8)`	Dieses Flag wird vom Areafiller benötigt.
`NOCROSSFILL,5 (-2Ø)`	Dieses Flag wird vom Areafiller benötigt.

I.4 Benutzer-Schnittstellen

Struktur IntuiMessage			
Struktur	ExecMessage	0	wird von Exec verwaltet
Langwort	Class	+20	Class der Meldung
Wort	Code	+24	Code der Meldung
Wort	Qualifier	+26	Qualifier der Meldung
Zeiger	IAddress	+28	
Wort	MouseX	+32	X-Koordinate der Maus
Wort	MouseY	+34	Y-Koordinate der Maus
Langwort	Seconds	+36	Sekunden
Langwort	Micros	+40	Mikro-Sekunden
Zeiger	IDCMPWindow	+44	Fenster für die Meldung
Zeiger	SpezialLink	+48	für System-Anwendung

ExecMessage
Diese Struktur, welche von Exec verwaltet wird, wird benötigt, um die Message in das System einzubinden und durch einen MessagePort auszugeben.

Class
Diese Variable korrespondiert direkt mit den IDCMP-Flags.

Code
Diese Variable enthält spezielle Werte wie Menü-Nummern. Dieses Daten-Feld hängt eng mit der Class-Variablen zusammen. Wenn Sie zum Beispiel die Maustasten abfragen, fragt Sie das Class-Feld ab, ob der Wert 8 anliegt (=MOUSEBUTTONS). Das Code-Feld fragt Sie nach dem Wert 104 (=SELECTDOWN).

Qualifier
Diese Variable enthält eine Kopie des ie__Qualifier-Feldes, welches von der Input Devicean die Intuition gesendet wurde. Sie können damit abfragen, ob die SHIFT- oder CTRL-Taste gedrückt wurde.

MouseX und MouseY
Hier finden Sie die Maus-Koordinaten als Abstand zur oberen linken Fensterecke.

Seconds and Micros
Diese beiden Langworte enthalten die Sekunden und die Mikrosekunden der System-Uhr.

Die IDCMP-Flags

Mit diesen Flags können Sie bzw. Ihr Programm bestimmen, welche Nachrichten Sie über den IDCMP empfangen wollen. Natürlich können Sie auch mehrere Flags gleichzeitig setzen. Sie müssen dann nur Ihre Abfrage entsprechend gestalten.

Fenster-Flags

NEWSIZE $2
Die Intuition sendet eine Meldung, nachdem der Anwender die Größe des Fensters verändert hat. Die neue Fenstergröße kann über die Größen-Variablen der Fenster-Struktur abgefragt werden.

REFRESHWINDOW $4
Es wird eine Meldung abgesetzt, wenn das Window erneuert (refresh) werden muß. Dieses Flag brauchen Sie nur setzen, wenn Sie Fenster vom Typ SIMPLE__REFRESH oder SMART__REFRESH einsetzen.

SIZEVERIFY $1
Mit diesem Flag weisen Sie die Intuition an, daß eine Zeichnung erst fertiggestellt sein muß, bevor der Anwender die Größe des Fensters verändern kann.

CLOSEWINDOW $200 (512)
Das Close-Gadget eines Fensters wurde betätigt. Das Fenster muß vom Programm geschlossen werden.

ACTIVEWINDOW $40000 (262144)
Es wird gemeldet, daß das Fenster aktiv ist.

INACTIVEWINDOW $80000 (524288)
Das abgefragte Fenster ist nicht aktiv.

Gadget-Flags

GADGETDOWN $20 (32)
Es wird das Drücken der Maustaste mit dem Pointer über einem Gadget gemeldet. Das Gadget muß das Flag GADGIMMEDIATE gesetzt haben.

GADGETUP $40 (64)
Diese Meldung wird abgesetzt, wenn nach einer Selektierung über einem Gadget die Taste wieder losgelassen wurde. In der Gadget-Struktur muß dafür das Flag REL-VERIFY gesetzt sein.

Requester-Flags

REQSET $80 (128)
Dieses Flag veranlaßt eine Meldung, wenn der erste Requester in einem Fenster erscheint.

REQCLEAR $1000 (4096)
Mit diesem Flag wird gemeldet, daß der letzte Requester eines Fensters verschwunden ist.

REQVERIFY $800 (2048)
Bei diesem Flag wartet die Intuition darauf, daß die Ausgabe eines anderen Requesters beendet wurde, bevor der neue Requester ausgegeben wird.

Menü-Flags

MENUPICK $100 (256)
Es erfolgt eine Meldung, daß ein Menüpunkt gewählt wurde. Die Menü-Nummer kann aus dem Code-Feld der Intuimessage geholt werden. Ist kein Menü selektiert, zeigt das Code-Feld auf 0.

MENUVERIFY $2000 (8192)
Mit diesem Flag weisen Sie die Intuition an, daß eine Zeichnung erst fertiggestellt sein muß, bevor der Anwender das Menü selektieren kann.

Maus-Flags

MOUSEBUTTONS $8
Es erfolgt eine Meldung, daß eine Maustaste gedrückt wurde. Über das Code-Feld der Intuimessage können Sie erfahren, welcher Knopf gedrückt oder losgelassen wurde. (SELECTDOWN=104, SELECTUP=232, *MENUDOWN, MENUUP*)

MOUSEMOVE $10 (16)
Wenn ebenfalls das Flag für REPORTMOUSE gesetzt ist, werden alle Mausbewegungen gemeldet.

DELTAMOVE $100000 (1048576)
Dieses Flag arbeitet mit dem Flag für MOUSEMOVE zusammen. Es wird die Entfernung zur letzten Position gemeldet.

Weitere Flags

VANILLAKEY $200000 (2097152)
Mit diesem Flag werden die RAWKEY-Ereignisse in ASCII-Codes übersetzt. Die Zeichen können dem Code-Feld der Intuimessage entnommen werden.

RAWKEY $400 (1024)
Unübersetzte Meldungen werden in das Code-Feld gesendet.

NEWPREFS $4000 (16384)
Wenn der Anwender die Preferences ändert, wird eine Meldung abgegeben.

DISKINSERTET $8000 (32768)
Das Einlegen einer Diskette wird mit einer Meldung quittiert.

DISKREMOVED $10000 (65536)
Mit diesem Flag wird die Entnahme einer Diskette registriert.

WBENCHMESSAGE $20000 (131072)

INTUITICKS $400000 (4194304)
Etwa 10 Mal in einer Sekunde kann der Timer abgefragt werden.

Struktur Gadget			
Zeiger	NextGadget	0	nächstes Gadget in der Liste
Wort	LeftEdge	4	linke Ecke im Ausgabe-Element
Wort	TopEdge	6	obere Ecke
Wort	Width	8	Breite des wählbaren Bereiches
Wort	Height	10	Höhe
Wort	Flags	12	Flags s.u.
Wort	Activation	14	Activation-Flags s.u.
Wort	GadgetType	16	Gadget-Typ s.u.
Zeiger	GadgetRender	18	Struktur Border oder Image
Zeiger	SelectRender	22	Struktur für selekt. Gadget
Zeiger	GadgetText	26	Text des Gadgets
Langwort	MutualExclude	30	zur Zeit ignoriert
Zeiger	SpecialInfo	34	auf PropInfo/StringInfo-Str.
Wort	GadgetID	38	
Zeiger	UserData	40	

Flags

GADGHCOMP $0	komplementiert die Bits in dem Auswahlrechteck
GADGHBOX $1	Rechteck um das Auswahlrechteck des Gadgets
GADGHIMAGE $2	alternatives Image oder eine andere Umrandung
GADGHNONE $3	keine Veränderung des Gadgets beim Selektieren
GADGIMAGE $4	Image, sonst als Border ausgegeben
GRELBOTTOM $8	TopEdge von Ober- oder Unterkante Ausgabe-Element
GRELRIGHT $10	(16) Lage von linker oder rechter Seite Ausgabe-Element
GRELWIDTH $20	(32) Breite nach Ausgabe-Element
GRELHEIGHT $40	(64) Höhe nach Ausgabe-Element
SELECTED $80	(128) Start als selektiertes Gadget
GADGDISABLED $100	(256) Zustand, aktiv/inaktiv.

Activation-Flags

RELVERIFY $1
Betätigung der Maustaste über dem Gadget wird erst dann registriert, wenn die Maustaste auch über dem Gadget wieder losgelassen wird.

GADGIMMEDIATE $2
reagiert sofort auf Mausklick.

TOGGLESELECT $100 (256)
schaltet zwischen normaler und selektierter Darstellung.

ENDGADGET $4
schließt Requester nach Selektierung des Gadgets.

FOLLOWMOUSE $8
Abfrage der Mausbewegung während der Selektierung.

RIGHTBORDER $10 (16)
setzt Gadget an die rechte Fensterbegrenzung.

LEFTBORDER $20 (32)
dto. für die linke Fensterseite.

TOPBORDER $40 (64)
dto. für die obere Grenze.

BOTTOMBORDER $80 (128)
dto. für die untere Grenze.

STRINGCENTER $200 (512)
zentriert den Text eines String-Gadgets in dem Auswahlfeld.

STRINGRIGHT $400 (1024)
gibt den Text eines String-Gadgets rechtsbündig zur Umrandung des Gadgets aus.

LONGINT $800 (2048)
Langwort eines String-Gadgets.

ALTKEYMAP $1000 (4096)
alternative Belegung der Tastatur.

GadgetType

BOOLGADGET $1	nach der Booleschen Wahrheitsregel nur zwei Zustände
PROBGADGET $3	Proportional-Gadget
STRGADGET $4	Typ String-Gadget
SCRGADGET $4000	(16384) Screen-Gadget
GZZGADGET $2000	(8192) für Gimmezerozero-Fenster
REQGADGET $1000	(4096) Requester-Gadget.

Struktur Requester			
Zeiger	OlderRequest	0	letztes Requester, vor diesem
Wort	LeftEdge	4	linke Ecke zu linke Ecke Fenster
Wort	TopEdge	6	obere Ecke zu obere Ecke Fenster
Wort	Width	8	Breite des Requester in Pixel
Wort	Height	10	Höhe des Requesters
Wort	RelLeft	12	links, Req.Pos. relativ zur Maus
Wort	RelTop	14	oben, Req.Pos. relativ zur Maus
Zeiger	ReqGadget	16	erstes Gadget des Requesters
Zeiger	ReqBorder	20	Z. auf Border-Str., wenn gew.
Zeiger	ReqText	24	Z. auf IntuiText des Requester
Wort	Flags	28	Flags s.u.
Byte	BackFill	30	Hintergrundfarbe des Requesters
Zeiger	ReqLayer	32	Layer Struktur für Requester
Struktur	ReqPad1 (32 Byte)		für System
Zeiger	ImageBMap		Anwender Bitmap
Zeiger	RWindow		für System
Struktur	ReqPad2 (36 Byte)		für System

Flags

POINTREL $1	Requester relativ zum Mauszeiger positionieren
PREDRAWN $2	eigene Bitmap-Struktur für den Requester
REQOFFWINDOW $1000	(4096) wenn Requester-Fenster inaktiv
REQACTIVE $2000	(8192) Requester ist gerade aktiv
SYSREQUEST $4000	(16384) System-Requester
DEFERREFRESH $8000	(32768) Req. unterbricht eine Refresh-Meldung.

Struktur StringInfo			
Zeiger	Buffer	0	Zeiger auf Start-Puffer
Zeiger	UndoBuffer	4	Zwischenspeicher/Eingabepuffer
Wort	BufferPos	8	Pufferposition/Cursor bei Start
Wort	MaxChars	10	max. Zeichenanzahl für Eingabe
Wort	DispPos	12	Pufferposition/erstes Zeichen
Wort	UndoPos	14	Zeichenposition/Zwischenspeicher
Wort	NumChars	16	Anzahl der Zeichen im Puffer
Wort	DispCount	18	Anzahl der sichtbaren Zeichen
Wort	CLeft	20	linke Kante der Box
Wort	CTop	22	obere Kante der Box
Zeiger	LayerPtr	24	Layer
Langwort	LongInt	28	lange Ganzzahl
Zeiger	AltKeyMap	32	alternativer Anwenderzeichensatz

I.5 Sprite-Strukturen

Struktur SimpleSprite			
Zeiger	posctldata	0	Sprite-Daten
Wort	height	4	Sprite-Höhe
Wort	x	6	Position-X
Wort	y	8	Position-Y
Wort	num	10	Spritenummer

Struktur userspritedata			
Wort	posctl	0	Eigenschaften, vom System vergeben
Wort	posctl	2	Eigenschaften, vom System vergeben
Wort	sprdata	4	1. Zeile, Bit-Ebene 0
Wort	sprdata	6	1. Zeile, Bit-Ebene 1
Wort	sprdata	8	2. Zeile, Bit-Ebene 0
Wort	sprdata	10	2. Zeile, Bit-Ebene 1
usw.			
Wort	reserved		Wert Null für nicht verknüpfte Hardware-Sprites

Struktur Mauszeiger			
Wort	posctl	0	Eigenschaften, vom System vergeben
Wort	posctl	2	Eigenschaften, vom System vergeben
Wort	sprdata	4	1. Zeile, Bit-Ebene 0
Wort	sprdata	6	1. Zeile, Bit-Ebene 1
Wort	sprdata	8	2. Zeile, Bit-Ebene 0
Wort	sprdata	10	2. Zeile, Bit-Ebene 1
usw.			
Wort	reserved		Wert Null für nicht verknüpfte Hardware-Sprites
Wort	reserved		Wert Null
Wort	reserved		Wert Null

I.6 Text-Strukturen

STRUKTUR TextAttr,0		
Zeige	Name	Zeiger auf den Namen der gewünschten Schriftart
Wort	YSize	Größe der gesuchten Schriftart
Byte	Style	Zeichenstil
Byte	Flags	preferences

Struktur des Avail-Puffers		
buffer	Wort	Anzahl der Schriftarten
b+2 1.font	Wort	flags 1=font v. Disk; 2=font v. Speicher
b+4	Langwort	Adresse des Namens vom Zeichensatz
b+8	Wort	y-Größe
b+10	Byte	Zeichenstil
b+11	Byte	preferences
b+12 2.font	Wort	flags
b+14	Langwort	etc.

Struktur IntuiText			
Byte	FrontPen	0	Vordergrundfarbe
Byte	BackPen	1	Hintergrundfarbe
Byte	DrawMode	2	Zeichenmodus
Wort	LeftEdge	4	linke Ecke, Textanfang
Wort	TopEdge	6	obere Ecke, Textanfang
Zeiger	TextAttr	8	Z. auf TextAttr-Struktur
Zeiger	IText	12	Z. auf Text-Zeichenkette
Zeiger	NextText	16	weitere IText-Struktur

I.7 Diverse Strukturen

Struktur Remember	
Länge	Bezeichnung
Zeiger	NextRemember
Langwort	RememberSize
Zeiger	Memory

Struktur Border			
Wort	LeftEdge	0	linke Ecke im Ausgabe-Element
Wort	TopEdge	2	obere Ecke im Ausgabe-Element
Byte	FrontPen	4	Vordergrundfarbe
Byte	BackPen	5	Hintergrundfarbe
Byte	DrawMode	6	Zeichenmodus
Byte	Count	7	Anzahl der Koordinaten-Paare
Zeiger	XY	8	Z. Koordinatenfeld
Zeiger	NextBorder	12	Z. weitere Border-Struktur

Struktur Image			
Wort	LeftEdge	0	linke Ecke im Ausgabe-Element
Wort	TopEdge	2	obere Ecke im Ausgabe-Element
Wort	Width	4	Breite der Image-Grafik
Wort	Height	6	Höhe der Grafik
Wort	Depth	8	Tiefe der Grafik
Zeiger	ImageData	10	Zeiger auf Daten für Grafik
Byte	PlanePick	14	
Byte	PlaneOnOff	15	
Zeiger	NextBorder	16	nächste Image-Struktur

Anhang II

Die Library-Routinen

Nachstehend finden Sie alle Library-Routinen, die in diesem Buch eingesetzt sind. Da eine Wertung nach wichtigen Befehlen und weniger wichtigen Routinen schwer möglich ist, sind die Libraries in alphabetischer Reihenfolge aufgeführt.

Alle Routinen sind nach einem einheitlichen Schema dargestellt. Zuerst finden Sie das Format der Routine und anschließend eine kurze Erklärung der einzelnen Parameter. Bei den Parametern finden Sie auch den dafür gültigen Typ der Variablen. Dabei ist zu beachten, daß die Zahlendarstellung beim Amiga-Basic nicht immer den Anforderungen der Variablen entspricht.

Basic-Variable	**Parameter der Routinen**
Integer (INT) lange Ganzzahl (LNG)	WORD, UWORD, BYTE STRUCT (Zeiger), CHAR (Zeiger), LONG

Probleme können bei den Integer-Werten auftreten. Da die kurze Ganzzahl beim Amiga-Basic einen Wert von –32768 bis 32767 annehmen kann, ein UWORD aber Werte von 0 bis 65535, muß der Wert unter Umständen korrigiert werden. Am besten ist dazu eine AND-Verknüpfung geeignet.

Beispiel: `POKEW Speicherstelle&, a% AND 65535`

Dieses Problem tritt selten auf. Sollten Sie jedoch einmal Schwierigkeiten damit bekommen, wissen Sie, wie Sie sie beheben können.

In der Kurzbeschreibung der einzelnen Routinen folgt noch eine knappe Erläuterung über die Funktion der Routine und ein ausgeschriebenes Beispiel. Aus dem Beispiel können Sie ebenfalls die Variablen-Typen der einzelnen Parameter entnehmen. Wenn es sich bei der Routine um eine Funktion handelt, ist außerdem das Feld *Ergebnis* ausgefüllt.

Die Routine selber muß mit einer Variablen, die in einer 4 Byte langen Speicherstelle abgelegt ist, aufgerufen werden. Mit diesem Aufruf wird eine Adresse zurückgegeben, und die braucht eben ein Langwort als Speicherstelle. Alle aufgeführten Routinen werden daher mit einer langen Ganzzahl (&) aufgerufen.

II.1 Die Diskfont-Library

Format: `fehler = AvailFonts(buffer,bufBytes,flags)`

Parameter:
buffer (LNG): Zeiger auf den reservierten Speicherbereich.
bufBytes (INT): Anzahl der Bytes im reservierten Speicher.
flags (INT): 1=AFF__DISK sucht die Fonts auf Diskette
2=AFF__MEMORY sucht die Fonts im Speicher.

Funktion: Es wird der reservierte Speicherbereich mit einem Header versehen und anschließend werden die Font-Daten eingetragen.

Beispiel: `fehl%=AvailFonts&(m&,5ØØ,3)`

Ergebnis: Wenn kein Fehler aufgetreten ist, wird 0 übergeben. Ist das Ergebnis <> 0, dann wird die Anzahl der benötigten Bytes mitgeteilt, damit der Puffer vergrößert werden kann.

Format: `font = OpenDiskFont(textAttr)`

Parameter: textAttr (LNG): Zeiger auf die TextAttr-Struktur, die von der Diskette geladen werden soll.

Funktion: Sucht auf der Diskette nach dem Zeichensatz, der *textAttr* entspricht, und liest ihn in den Speicher.

Beispiel: `font&=OpenDiskFont&(VARPTR(ta&(Ø)))`

Ergebnis: Bei einem Fehler wird 0 übergeben, sonst ein Zeiger auf die Font-Struktur, die zum Setzen des Zeichensatzes benötigt wird.

II.2 Die DOS-Library

Format: `Close(file)`

Parameter: file (LNG): Adresse der FileHandle-Struktur, die bei *Open* übergeben wurde.

Funktion: Es wird eine geöffnete Datei geschlossen.

Beispiel: `CALL xClose&(v&)`

Ergebnis: keines.

Format: `file = Open(name,accessMode)`

Parameter: name (LNG): Adresse eines mit 0 abgeschlossenen Strings.
accessMode (INT): Befehls-Modus.
MODE_OLDFILE (1005) – existierende Datei lesen/schreiben.
MODE_NEWFILE (1006) – neue Datei für die Eingabe.

Funktion: Es wird die Datei *name* für die Ein- oder Ausgabe geöffnet.

Beispiel: `v&=xOpen&(SADD(fenster-),1006)`

Ergebnis: Zeiger auf die FileHandle-Struktur oder 0, wenn diese Datei nicht geöffnet werden konnte.

II.3 Die Exec-Library

Format: `memoryBlock = AllocMem(byteSize,requirements)`

Parameter: byteSize: zu reservierender Speicherbereich in Bytes.
requirements (LNG):
MEMF_PUBLIC = 2^0 – Speicherbereich nicht verschiebbar.
MEMF_CHIP = 2^1 – für Grafik-Chips unter 512 Kbyte.
MEMF_FAST = 2^2 – außer Reichweite der Spez.Chips > 512 Kbyte.
MEMF_CLEAR = 2^16 – res. Speicher mit Null-Bits belegt.

Funktion: Es wird ein Speicherbereich nach den Anforderungen zugeteilt.

Beispiel: `memblock&=AllocMem&(800,65537&)`

Ergebnis: Die Funktion gibt den Wert 0 zurück, wenn der Speicher zu klein ist, sonst einen Zeiger auf den Memory-Block.

Format: `FreeMem(memoryBlock,byteSize)`

Parameter: memoryBlock (LNG): Zeiger auf den reservierten Memory-Block.
byteSize: reservierter Speicher in Byte.

Funktion: Ein reservierter Speicherbereich wird an das System zurückgegeben.

Beispiel: `CALL FreeMem&(mem&,8)`

Ergebnis: keines.

Format: `message = GetMsg(port)`

Parameter: port (LNG): Zeiger auf den MessagePort.

Funktion: Diese Funktion holt eine Nachricht von dem MessagePort.

Beispiel: `mess&=GetMsg&(me&)`

Ergebnis: Zeiger auf die zuerst verfügbare Nachricht, ist das Ergebnis Null, so liegt keine Nachricht vor.

Format: `ReplyMsg(message)`

Parameter: message (LNG): Zeiger auf die Nachricht, welche mit *GetMsg* geholt wurde.

Funktion: Es wird die Nachricht an ihren ReplyPort zurückgesandt, so daß sie neu verwendet werden kann.

Beispiel: `CALL ReplyMsg&(mess&)`

Ergebnis: keines.

II.4 Die Grafik-Library

Format: `plane = AllocRaster(width,height)`

Parameter: width (INT): Breite der Bit-Ebene in Bits.
height (INT): Höhe der Bit-Ebene in Bits.

Funktion: Es wird für eine Bit-Ebene ein Speicherbereich zugeteilt.

Beispiel: `bp&=AllocRaster&(1008,512)`

Ergebnis: Reicht die vorhandene Speicherkapazität nicht aus, wird 0 zurückgegeben, andernfalls ein Zeiger auf den Memory-Block.

Format: `BltClear(memBlock,bytecount,flags)`

Parameter: memBlock (LNG): Memory-Block, der gelöscht werden soll.
bytecount (LNG): Anzahl der Bytes, die gelöscht werden sollen.
flags (Byte): Mit dem Wert 0 werden die Bytes auf Null gesetzt.

Funktion: Es wird der Speicher-Block gelöscht.

Beispiel: `CALL BltClear&(bp&(i),64512&,Ø)`

Ergebnis: keines.

Format: `ChangeSprite(vp,s,newdata)`

Parameter: vp (LNG): Zeiger auf die ViewPort-Struktur des Sprites. Bei einem Wert von Null gilt der aktuelle ViewPort.
s (LNG): Zeiger auf die SimpleSprite-Struktur.
newdata (LNG): Zeiger auf die neue Datenstruktur.

Funktion: Die Darstellung des Sprites wird durch ein neues Datenfeld geändert.

Beispiel: `CALL ChangeSprite&(Ø,m1&,m2&)`

Ergebnis: keines.

Format: `ClearScreen(rastPort)`

Parameter: rastPort (LNG): zeigt auf die RastPort-Struktur.

Funktion: Ab der aktuellen Position des Grafik-Cursors wird der Bildschirm gelöscht bzw. auf Null gesetzt. Im Zeichnungs-Modus 2 wird der Screen auf die Hintergrundfarbe gesetzt.

Beispiel: `CALL ClearScreen&(rp&)`

Ergebnis: keines.

Format: `ClipBlit`
`ClipBlit(Src,SrcX,SrcY,Dest,DestX,DestY,XSize,YSize,Minterm)`

Parameter: Src (LNG): Zeiger auf den RastPort, von dem aus der rechteckige Bereich kopiert wird.
SrcX (INT), SrcY (INT): Koordinaten der oberen linken Ecke des zu übertragenden Rechteckbereiches des Quell-RastPorts.

Dest (LNG): Dieser Ziel-RastPort kann der gleiche RastPort der Quelle sein, oder irgendein anderer RastPort.
DestX (INT), DestY (INT): Die Koordinaten der oberen linken Ecke des Rechteckes in dem Ziel-RastPort, auf den das Rechteck übertragen wird.
XSize (INT), YSize (INT): Größe des Rechteckes (in Pixel), welches kopiert wird.
Minterm: Während des Kopiervorganges können 256 (1 Byte) verschiedene Verknüpfungen, abhängig von den vier DMA-Kanälen, durchgeführt werden. Für uns sind dabei folgende Werte interessant:
$30 Das ursprüngliche Rechteck wird invertiert und dann kopiert.
$50 Das ursprüngliche Rechteck wird kopiert und am Ziel invertiert.
$C0 Es wird alles kopiert.

Funktion: Die Routine verschiebt bzw. kopiert einen rechteckigen Bereich aus dem ursprünglichen RastPort an eine andere Position des Quell-RastPorts oder in einen anderen RastPort.

Beispiel: `CALL ClipBlit&(rp1&,x%,y%,rp2&,b%,h%,192)`

Ergebnis: keines.

Format: `CloseFont(textFont)`

Parameter: textFont (LNG): Zeiger auf die TextFont-Struktur, die mit OpenFont übergeben wurde.

Funktion: Es wird der aktuelle Zeichensatz geschlossen und der benötigte Speicherplatz wieder freigegeben.

Beispiel: `CALL CloseFont&(font&)`

Ergebnis: keines.

Format: `CBump(c)`

Parameter: c (LNG): Zeiger auf die UserCopperList.

Funktion: Es wird veranlaßt, daß in der CopperListe eine Position weitergegangen wird.

Beispiel: `CALL CBump&(uc&)`

Ergebnis: keines.

Format: `CMOVE(c,a,v)`

Parameter: c (LNG): Zeiger auf die UserCopperList.
a (INT): Hardware-Register.
v (INT): was ins Hardware-Register geschrieben wird.

Funktion: Es wird eine MOVE-Anweisung in die UserCopperList geschrieben.

Beispiel: `CALL CMove&(uc&,384,&HØ)`

Ergebnis: keines.

Format: `CWAIT(c,v,h)`

Parameter: c (LNG): Zeiger auf die UserCopperList.
v (INT): vertikale Position des Elektronenstrahls.
h (INT): horizontale Position des Elektronenstrahls.

Funktion: Es wird eine WAIT-Anweisung in die UserCopperList geschrieben.

Beispiel: `CALL CWait&(uc&,a1,1Ø)`

Ergebnis: keines.

Format: `Draw(rastPort,x,y)`

Parameter: rastPort (LNG): Zeiger auf die RastPort-Struktur.
x (INT): X-Koordinate in Pixel.
y (INT): Y-Koordinate in Bildpunkten.

Funktion: Diese Routine zieht eine Linie vom grafischen Cursor zu der durch x und y festgelegten Position.

Beispiel: `CALL Draw& (WINDOW(8),5Ø,8Ø)`

Ergebnis: keines.

Format: `DrawEllipse(rastPort,x,y,hr,vr)`

Parameter: rastPort (LNG): Zeiger auf die RastPort-Struktur.
x (INT): X-Koordinate des Mittelpunktes in Pixel.
y (INT): Y-Koordinate in Bildpunkten.
hr (INT): horizontaler Radius.
vr (INT): vertikaler Radius.

Funktion: Diese Routine zeichnet eine Ellipse.

Beispiel: `CALL DrawEllipse&(rp&,50,80,20,10)`

Ergebnis: keines.

Format: `Flood(rp,mode,x,y)`

Parameter: rp (LNG) zeigt auf den Rastport, z.B WINDOW(8).
mode spezifiziert den Füllmodus 0 oder 1, wie unten beschrieben.
x (INT), y (INT): Diese beiden Variablen enthalten die Koordinaten des Grafikpunktes, mit dem das Füllen begonnen wird.

Funktion: Es wird eine bestimmte Fläche farbig ausgemalt.

Beispiel: `CALL Flood&(rp&,1,x%,y%)`

Ergebnis: keines.

mode 0: outline mode
In diesem Modus sucht das System, beginnend von dem vorgegebenen Grafikpunkt, in allen Richtungen nach einem Pixel, welcher der Farbe des AOL-Pens (Randfarbe) entspricht. Diese Fläche wird dann gefüllt, wobei alle andersfarbigen Pixel übermalt werden.

mode 1: color mode
Dabei wird der Pixel an den spezifizierten Koordinaten für die Farbgebung herangezogen. Alle Grafikpunkte, welche die gleiche Farbe haben, werden gefüllt.

Format: `FreeRaster(p,width,height)`

Parameter: p (LNG) zeigt auf den reservierten Speicherbereich.
width (INT) ist die Breite der BitPlane in Bits.
height (INT) ist die Höhe der Bit-Ebene in Bits.

Funktion: Der reservierte Speicherbereich wird freigegeben.

Beispiel: `CALL FreeRaster&(bp&(i),1008,512)`

Ergebnis: keines.

Format: `FreeSprite(num)`

Parameter: num (INT) ist die Nummer des Sprites, das zurückgegeben wird.

Funktion: Das Sprite wird zur neuen Verwendung freigegeben.

Beispiel: `CALL FreeSprite&(num)`

Ergebnis: keines.

Format: `Farbe = GetRGB4(colormap,entry)`

Parameter: colormap (LNG) zeigt auf die Colormap-Struktur.
entry (INT) zeigt mit der Farbnummer auf das Wort mit den Farbwerten in der Colormap.

Funktion: Der Farbwert wird aus der Farbtafel geholt.

Beispiel: `farb%= GetRGB4&(cm&,i%)`

Ergebnis: Wenn eine ungültige Farbnummer angegeben wird, ist das Ergebis –1, sonst wird der RGB-Wert in Wortlänge übergeben. Die einzelnen Farbwerte sind in jeweils 4 Bit, von rechts beginnend, enthalten.

Format: `Nummer=GetSprite(sprite,pick)`

Parameter: sprite (LNG): Dieses ist ein Zeiger auf die SimpleSprite-Struktur.
pick (INT): Die Nummer des gewünschten Sprites zeigt dieser Parameter.

Funktion: Eines der 8 Sprites (0–7) wird zugeteilt.

Beispiel: `num&=GetSprite&(m1&,2)`

Ergebnis: Die Funktion liefert die Nummer des zugeteilten Sprites oder –1.

Bei dieser Funktion können Sie die Nummer eines von Ihnen gewünschten Sprites belegen oder sich das nächste freie Sprite zuteilen lassen. Wollen Sie ein bestimmtes Sprite belegen, so geben Sie einfach diese Nummer unter pick ein. Die Funktion liefert dann –1, wenn dieses Sprite ordnungsgemäß belegt werden konnte. Überlassen Sie jedoch die Belegung eines Sprites der Funktion, so geben Sie unter pick –1 ein. Das Ergebnis der Funktion ist dann die Nummer des Sprites, welches belegt wurde. Ist kein Sprite verfügbar, so liefert die Funktion den Wert –1.

Format: `InitBitMap(bm,depth,width,height)`

Parameter: bm (LNG) zeigt auf eine Bitmap-Struktur.
depth (INT) enthält die Anzahl der Bit-Ebenen der Bitmap.
width (INT): Breite der Bitmap in Bits.
height (INT) enthält die Höhe der Bitmap in Bits.

Funktion: Es werden diverse Elemente der Bitmap-Struktur initialisiert.

Beispiel: `CALL InitBitMap&(mb&,2,1ØØ8,512)`

Ergebnis: keines.

Format: `InitTmpRas(tmpras,buffer,size)`

Parameter: tmpras (LNG) zeigt auf eine TmpRas-Struktur.
buffer (LNG) zeigt auf den reservierten Speicherbereich.
size (INT): Größe des reservierten Puffers.

Funktion: Es wird ein Speicherbereich für Flood, Area etc. initialisiert.

Beispiel: `CALL InitTmpRas&(mb&,bp&,volum)`

Ergebnis: keines.

Format: `InitRastPort(rp)`

Parameter: rp (LNG) zeigt auf eine RastPort-Struktur bzw. auf einen dafür reservierten Speicherbereich.

Funktion: Es wird eine RastPort-Struktur mit den Standard-Werten initialisiert.

Beispiel: `CALL InitRastPort&(rp&)`

Ergebnis: keines.

Format: `LoadRGB4(viewPort,colormap,count)`

Parameter: viewPort (LNG) zeigt auf den ViewPort, dessen Farben geändert werden sollen.
colormap (LNG) zeigt auf die Tafel mit RGB-Werten aus kurzen Ganzzahlen. Dabei wird die Intensität von 0 als Minimum bis 15 als Maximum eingegeben.

Hintergrund	– 0RGB
Farbe 1	– 0RGB
Farbe 2	– 0RGB etc.
count (INT):	Die Anzahl der Farben in der Farbtafel.

Funktion: Die Farben werden in die Farbtafel des ViewPorts abgelegt.

Beispiel: `CALL LoadRGB4&(vp&,VARPTR(farbe%(Ø)),4)`

Ergebnis: keines.

Format: `Move(rastPort,x,y)`

Parameter: rastPort (LNG) zeigt auf den Rastport (z.B. WINDOW(8)).
x (INT): Anzahl der Pixel vom linken Rand.
y: Den Abstand zum oberen Fensterrand.

Funktion: Mit der Routine wird der grafische Cursor auf die durch x und y festgelegte Position gesetzt.

Beispiel: `CALL Move& (WINDOW(8),5,8)`

Ergebnis: keines.

Format: `MoveSprite(vp,sprite,x,y)`

Parameter: vp (LNG): Dieses ist ein Zeiger auf die ViewPort-Struktur des Sprites. Bei einem Wert von Null gilt der aktuelle ViewPort.
sprite (LNG): Zeiger auf die SimpleSprite-Struktur.
x (INT), y(INT): neue Sprite-Position.

Funktion: Das Bild des Sprites wird auf die neue Position bewegt.

Beispiel: `CALL MoveSprite&(Ø,m1&,34,1ØØ)`

Ergebnis: keines.

Format: `OpenFont(textAttr)`

Parameter: textAttr (LNG) zeigt auf eine TextAttr-Struktur mit den Eigenschaften des Zeichenstils.

Funktion: Es wird im System nach dem Standard-Zeichensatz gesucht, welcher den Eigenschaften entspricht.

Beispiel: `font&=OpenFont&(VARPTR(ta&(Ø)))`

Ergebnis: Bei einem Fehler wird 0 zurückgegeben, sonst ein Zeiger auf die Font-Struktur, welche zum Setzen des Zeichensatzes benötigt wird.

Format: `PolyDraw(rastPort,count,array)`

Parameter: rastPort (LNG) ist der Zeiger auf den Rastport, z.B. WINDOW(8).
count (INT): Anzahl der Koordinaten-Paare.
array (LNG): Adresse der ersten Feldelementes.

Funktion: Es werden die einzelnen Koordinaten-Paare durch Linien miteinander verbunden.

Beispiel: `CALL PolyDraw&(rp&,1Ø3,VARPTR(amiga(Ø)))`

Ergebnis: keines.

Format: `farbe=ReadPixel(rastPort,x,y)`

Parameter: rastPort (LNG) zeigt auf den Rastport, z.B. WINDOW(8).
x (INT): Position in Bildpunkte vom linken Fensterrand.
y (INT): Abstand vom oberen Fensterrand.

Funktion: Es wird die Farbkennung des Bildpunktes bei der Koordinate x,y im aktuellen Fenster geliefert.

Beispiel: `vorh%= ReadPixel&(WINDOW(8),x,5Ø)`

Ergebnis: Der Farbwert an der spezifizierten Koordinate wird zurückgegeben.

Format: `RectFill(rastPort,xl,yl,xu,yu)`

Parameter: rastPort (LNG): zeigt auf den Rastport, z.B. WINDOW(8).
xl (INT) ist die obere linke x-Koordinate und
yl (INT) die obere linke y-Koordinate des Rechteckes.
xu (INT): untere rechte x-Koordinate.
yu (INT): untere rechte y-Koordinate des Rechteckes.

Funktion: Die Routine füllt einen rechteckigen Bereich mit der aktuellen Vordergrundfarbe bzw. einem Muster der PATTERN-Anweisung in dem gesetzten Zeichen-Modus.

Beispiel: `CALL RectFill& (rp&,x1,y1,x2,y2)`

Ergebnis: keines.

Format: `ScrollVPort(vP)`

Parameter: vP (LNG): Zeiger auf den Viewport, der gescrollt werden soll.

Funktion: Der komplette Bildschirmbereich wird gescrollt.

Beispiel: `CALL scrollVPort& (vp&)`

Ergebnis: keines.

Format: `SetAPen(rastPort,pen)`

Parameter: rastPort (LNG): Zeiger auf den Rastport, z.B. WINDOW(8).
pen (INT): Farbe.

Funktion: Es wird die Vordergrundfarbe für Text- und grafische Anweisungen gesetzt.

Beispiel: `CALL SetAPen& (WINDOW(8),3)`

Ergebnis: keines.

Format: `SetBPen(rastPort,pen)`

Parameter: rastPort (LNG): Zeiger auf den Rastport, z.B. WINDOW(8).
pen (INT): Farbe.

Funktion: Die Hintergrundfarbe für Text- und grafische Anweisungen auf dem Bildschirm wird gesetzt.

Beispiel: `CALL SetBPen& (WINDOW(8),1)`

Ergebnis: keines.

Format: `SetDrMd(rastPort,drawMode)`

Parameter: rastPort (LNG): zeigt auf den Rastport, z.B. WINDOW(8).
drawMode (INT): Die einzelnen Modi sind oben erklärt.

Funktion: Die Routine setzt für die folgenden grafischen Anweisungen den Zeichenmodus fest.

Beispiel: `CALL SetDrMd& (WINDOW(8),3)`

Ergebnis: keines.

Format: `Fehler=SetFont(RastPort,font)`

Parameter: RastPort (LNG): Rastport, z.B. WINDOW(8).
font (LNG) zeigt auf Struktur eines geöffneten Zeichensatzes.

Funktion: Die Routine setzt den Zeichensatz in die Rastport-Struktur und korrigiert die Ausgabe-Parameter.

Beispiel: `Fehler=SetFont&(WINDOW(8),font&)`

Library: graphics.library

Ergebnis: Wenn kein Fehler registriert wurde, wird 0 ausgegeben.

Format: `SetRast(rastPort,color)`

Parameter: rastPort (LNG) zeigt auf die Rastport-Struktur, z.B. WINDOW(8).
color (INT) ist die Farbnummer, mit der gefärbt wird.

Funktion: Der komplette Bildschirm wird in einer wählbaren Farbe ausgegeben.

Ergebnis: keines.

Format: `SetRGB4(viewPort,index,r,g,b)`

Parameter: viewPort (LNG): Zeiger auf die Viewport-Struktur.
index (INT): Farbnummer, die gesetzt werden soll.
r,g,b (INT): Der Rot-, Grün- und Blauanteil mit einem Wert von 0 bis 15.

Funktion: Der Farbton einer Farbe wird aus den festgelegten RGB-Anteilen gesetzt.

Beispiel: `CALL SetRGB4&(vp&,5,15,7,4)`

Ergebnis: keines.

Format: `Fehler=Text(RastPort,string,count)`

Parameter: rastPort (LNG) zeigt auf die Rastport-Struktur, z.B. WINDOW(8).
string (LNG) zeigt auf die Anfangsadresse einer Zeichenkette.
count (INT) gibt die Länge des auszugebenden Textes an.

Funktion: Der linke Teil oder eine komplette Zeichenkette wird ausgegeben.

Beispiel: `Fehler=Text&(WINDOW(8),v&,12)`

Ergebnis: Wenn bei der Ausgabe kein Fehler auftrat, wird von der Funktion 0 zurückgegeben.

Format: `WritePixel(rastPort,x,y)`

Parameter: rastPort (LNG): Zeiger auf die RastPort-Struktur (WINDOW(8)).
x (INT): Abstand in Pixel vom linken Rand.
y (INT): Anzahl der Bildpunkte vom oberen Rand gemessen.

Funktion: Es wird ein Bildpunkt bei der Koordinate x,y gesetzt.

Beispiel: `erg=WritePixel& (WINDOW(8),1ØØ,5Ø)`

Ergebnis: Werden die Grenzen des Rastports überschritten, wird –1 zurückgegeben, sonst 0.

II.5 Die Intuition-Library

Format: `CloseScreen(Screen)`

Parameter: Screen (LNG) zeigt auf eine Screen-Struktur.

Funktion: Ein geöffneter Intuition-Screen wird geschlossen.

Beispiel: `CALL CloseScreen&(scr&)`

Ergebnis: keines.

Format: `MoveScreen(Screen,DeltaX,DeltaY)`

Parameter: Screen (LNG): zeigt auf eine Screen-Struktur.
DeltaX: Es ist keine horizontale Bewegung möglich.
DeltaY (INT): Anzahl der Zeilen um die in Y-Richtung gescrollt werden soll.

Funktion: Der spezifizierte Bildschirm wird in vertikaler Richtung bewegt.

Beispiel: `CALL moveScreen&(scr&,Ø,1)`

Ergebnis: keines.

Format: `screen=OpenScreen(NewScreen)`

Parameter: NewScreen (LNG) zeigt auf eine NewScreen-Struktur mit den Eigenschaften des neuen Bildschirmes.

Funktion: Es wird ein neuer Intuition-Screen geöffnet, der den spezifizierten Eigenschaften entspricht.

Beispiel: `scr&= OpenScreen&(m&)`

Ergebnis: Die Funktion liefert bei einem Fehler 0, sonst einen Zeiger auf den neuen Screen.

Format: `window=OpenWindow(NewWindow)`

Parameter: NewWindow (LNG) zeigt auf die NewWindow-Struktur des Fensters, welches geöffnet werden soll.

Funktion: Es wird ein neues Fenster geöffnet, das den spezifizierten Eigenschaften der Struktur entspricht.

Beispiel: `win&= OpenWindow&(mem&)`

Ergebnis: Die Funktion liefert bei einem Fehler 0, sonst einen Zeiger auf die neue Window-Struktur.

Format: `CloseWindow(Window)`

Parameter: Window (LNG) zeigt auf die Window-Struktur des Fensters, das geschlossen werden soll.

Funktion: Ein geöffnetes Intuition-Window wird geschlossen.

Beispiel: `CALL CloseWindow&(win&)`

Ergebnis: keines.

Format: `SizeWindow(Window,DeltaX,DeltaY)`

Parameter: Window (LNG) zeigt auf die Window-Struktur des Fensters, das in der Größe verändert werden soll.
DeltaX (INT): Wert für die Vergrößerung oder Verkleinerung in der X-Achse.
DeltaY (INT): Anzahl der Zeilen, um die in Y-Richtung die Größe verändert werden soll.

Funktion: Das angesprochene Fenster wird in der Größe verändert.

Beispiel: `CALL SizeWindow&(win&,8,4)`

Ergebnis: keines.

Format: `SetWindowTitles(Window,WindowTitle,ScreenTitle)`

Parameter: Window (LNG) zeigt auf die Window-Struktur des Fensters, dessen Titel geändert werden soll.
WindowTitle (LNG) zeigt auf einen Text-String, der mit 0 abgeschlossen sein muß. Es können aber auch die Werte 0 und –1 gesetzt werden.
ScreenTitle zeigt auf den Text-Sting für den Screen-Titel. Wahlweise kann auch ein Wert von 0 oder –1 eingegeben werden.

Funktion: Der Titel des angesprochenen Fensters oder Screens erhält einen neuen Text.

Beispiel: `CALL SetWindowTitles&(WINDOW(7),ti&,-1)`

Ergebnis: keines.

Format: `Fehler=WindowLimits(Window,MinWidth,MinHeight,MaxWidth,MaxHeight)`

Parameter: Window (LNG) zeigt auf die Window-Struktur des Fensters, dessen Größenbeschränkungen geändert werden sollen.
MinWidth (INT) setzt die minimale Breite des Fensters.
MinHeight (INT) setzt die minimale Höhe des Windows.
MaxWidth (INT) setzt die maximale Breite und
MaxHeight (INT) ist die maximale Höhe des Fensters.

Funktion: Die maximalen und die minimalen Fenstergrenzen werden neu festgelegt.

Beispiel: `fe&=WindowLimits&win&,1ØØ,4Ø,Ø,18Ø)`

Ergebnis: Wenn *wahr* zurückgegeben wird, konnten die neuen Werte übernommen werden. Überschreiten die Minimalwerte die aktuelle Fenstergröße, wird unwahr zurückgegeben. Bei den Maximalwerten verhält es sich ebenso. Ein Wert von 0 bewirkt, daß dieser Wert nicht geändert wird.

Format:
```
wahr=AutoRequest(Window,BodyText,PositiveText,NegativeText,
**              PositiveFlags,NegativeFlags,Width,Height)
```

Parameter: Window (LNG): Struktur des Windows, zu dem der Requester gehört.
BodyText (LNG) zeigt auf IntuiText-Struktur für die Erläuterungen.
PositiveText (LNG) zeigt auf IntuiText für die positive Antwort.
NegativeText (LNG) zeigt auf Struktur für die negative Antwort.

PositiveFlags (INT) erhält Flags für den IDCMP für externe Ereignisse, die der Nachfrage genügen.
NegativeFlags (INT) sind IDCMP-Flags für negative Ereignisse.
Width (INT): Breite des Requesters.
Height (INT): Höhe des Requesters.

Funktion: Es wird ein Requester mit Bool-Gadgets konstruiert und auf dem Bildschirm ausgegeben. Je nach Selektierung oder anderen Ereignissen (über den IDCMP) wird wahr oder unwahr zurückgegeben.

Beispiel: `jn&=AutoRequest&(WINDOW(7),m&,m&+6Ø,m&+8Ø,Ø,Ø,3ØØ,7Ø)`

Ergebnis: Null wird zurückgegeben, wenn das Negativ-Gadget selektiert, oder –1, wenn das Positiv-Gadget gewählt wurde.

Format: `Requester=Request(Requester,Window)`

Parameter: Requester (LNG): zeigt auf die Requester-Struktur, die ausgegeben werden soll.
Window (LNG): Dies ist ein Zeiger auf die Fenster-Struktur, in die der Requester gesetzt wird, z.B. WINDOW(7).

Funktion: Es wird der spezifizierte Requester auf dem Bildschirm ausgegeben.

Beispiel: `rq&= Request&(mr&,w&)`

Ergebnis: Es wird Null zurückgegeben, wenn dieser Requester nicht geöffnet werden konnte, oder –1, wenn der Requester geöffnet ist.

Format: `EndRequest(Requester,Window)`

Parameter: Requester (LNG) zeigt auf die Requester-Struktur.
Window (LNG): Zeiger auf die Fenster-Struktur, aus welcher der Requester entfernt wird.

Funktion: Es wird der Requester aus dem Fenster und der Window-Struktur entfernt.

Beispiel: `CALL EndRequest&(mr&,w&)`

Ergebnis: keines.

Format: `SetPointer(Window,Pointer,Height,Width,XOffset,YOffset)`

Parameter: Window (LNG) zeigt auf die Window-Struktur.
Pointer (LNG) zeigt auf das Datenfeld des Sprites.
Height (INT) erhält die Höhe des Sprites in Grafikzeilen.
Width (INT) erhält die Breite des Sprites in Pixel (< =16).
XOffset (INT) erhält den Abstand in X-Richtung und
YOffset (INT) den Abstand in Y-Richtung von der idealen
Pointer-Position.

Funktion: Es wird ein Anwender-Mauszeiger ausgegeben.

Beispiel: `CALL SetPointer&(WINDOW(7),m2&,9,16,1,1)`

Ergebnis: keines.

Format: `ClearPointer(Window)`

Parameter: Window (LNG) zeigt auf die Window-Struktur.

Funktion: Es wird ein Anwender-Mauszeiger aus der Window-Struktur gelöscht.

Beispiel: `CALL ClearPointer&(WINDOW(7))`

Ergebnis: keines.

Format: `gadget=AddGadget(Pointer,Gadget,Position)`

Parameter: Pointer (LNG) zeigt auf das Fenster, zu dem das Gadget gehört.
Gadget (LNG) zeigt auf die Struktur des neuen Gadgets.
Position ist die Position des neuen Gadgets in der Gadget-Liste.

Funktion: Fügt das neue Gadget in die Gadget-Liste des aktuellen Fensters ein.

Beispiel: `gad= AddGadget&(WINDOW(7),mem&,-1)`

Ergebnis: Die aktuelle Position in der Gadget-Liste wird zurückgegeben.

Format: `Fehler=RemoveGadget(Pointer,Gadget)`

Parameter: Pointer (LNG) zeigt auf die Window-Struktur des Fensters, aus dem das Gadget entfernt werden soll.
Gadget (LNG) zeigt auf die Struktur des Gadgets, welches entfernt wird.

Funktion: Das Gadget wird aus der Gadget-Liste des aktuellen Fensters entfernt.

Beispiel: `rgad&= RemoveGadget&(WINDOW(7),mem&)`

Ergebnis: Die Funktion gibt die Position, unter der das Gadget in der Liste war, zurück, oder –1, wenn das Gadget nicht gefunden wurde.

Format: `OnGadget(Gadget,Pointer,Requester)`

Parameter: Gadget (LNG) zeigt auf die Struktur des zu aktivierenden Gadgets. Pointer (LNG) zeigt auf die Window-Struktur, zu der das Gadget gehört. Requester (LNG) zeigt auf die Requester-Struktur, zu der das Gadget gehört. Wenn das Gadget kein Requester-Gadget ist, wird 0 eingegeben.

Funktion: Das angegebene Gadget wird aktiviert.

Beispiel: `CALL OnGadget&(mem&,WINDOW(7),Ø)`

Ergebnis: keines.

Format: `RethinkDisplay`

Parameter: keine.

Funktion: Die kompletten Parameter für die Bildausgabe werden überprüft und richtiggestellt.

Beispiel: `CALL RethinkDisplay&`

Ergebnis: keines.

Format: `RemakeDisplay`

Parameter: keine.

Funktion: Die kompletten Parameter für die Bildausgabe werden überprüft und richtiggestellt. Erweiterte Funktion von RethinkDisplay.

Beispiel: `CALL RemakeDisplay&`

Ergebnis: keines.

Format: `DrawImage(RastPort,Image,LeftOffset,TopOffset)`

Parameter: RastPort (LNG) zeigt auf die RastPort-Struktur, z.B. WINDOW(8).
Border (LNG) enthält den Zeiger auf die Image-Struktur.
LeftOffset (INT): Abstand in Pixel, der zur X- und
TopOffset (INT): Abstand, der zur Y-Koordinate von Image addiert wird.

Funktion: Zeichnet ein Grafikbild mit den Daten, die in der Image-Struktur festgelegt sind.

Beispiel: `CALL DrawImage&(WINDOW(8),im&,25Ø,1ØØ)`

Ergebnis: keines.

Format: `v=ViewAddress`

Parameter: keine.

Funktion: Es wird die Adresse der Intuition View-Struktur übergeben.

Beispiel: `v&=ViewAddress&`

Ergebnis: Die Funktion übergibt die Adresse der View-Struktur.

Format: `vp=ViewPortAddress(Window)`

Parameter: Window (LNG) zeigt auf eine Window-Struktur.

Funktion: Es wird die Adresse der Intuition ViewPort-Struktur übergeben.

Beispiel: `vp&=ViewPortAddress&(WINDOW(7))`

Ergebnis: Die Funktion übergibt die Adresse der Viewport-Struktur.

Format: `Speicher=AllocRemember(RememberKey,Size,Flags)`

Parameter: RememberKey (LNG) zeigt auf eine Remember-Struktur. Vor dem ersten Aufruf muß dieser Zeiger auf Null gesetzt werden.
Size (INT) enthält die Größe des belegten Speichers in Bytes.
Flags (LNG): Die Art der Speicherbelegung (siehe AllocMem) wird hiermit gesetzt.

Funktion: Es wird die Exec-Routine Exec AllocMem aufgerufen. Die Parameter werden in einer speziellen Liste hinterlegt.

Beispiel: `meb&=AllocRemember&(Ø,5ØØ,art&)`

Ergebnis: Bei einem Fehler wird 0 übergeben, sonst ein Zeiger auf den reservierten Speicherblock.

Format: `FreeRemember(RememberKey,ReallyForget)`

Parameter: RememberKey (LNG) zeigt auf eine Remember-Struktur.
ReallyForget: Wenn der Wert dieser Variablen logisch wahr (–1) ist, werden alle reservierten Speicherbereiche freigegeben.

Funktion: Es werden die mit AllocRemember belegten Bereiche freigegeben.

Beispiel: `CALL FreeRemember&(rk&,-1)`

Ergebnis: keines.

II.6 Die Layers-Library

Format: `Layer=CreateUpFrontLayer(li,bm,xØ,yØ,x1,y1,flags,±bm2[)`

---

li (LNG): Zeiger auf eine LayerInfo-Struktur.
bm (LNG) zeigt auf die Bitmap, die von allen anderen Layers benutzt wird. Damit ist die normale Bitmap der Video-Ausgabe gemeint.
x0 (INT) und *y0* (INT): linke obere Ecke des Layers.
x1 (INT) und *y1* (INT): rechte untere Ecke des Layers.
flags: Für die verschiedenen Layer-Typen wird ein Bit gesetzt.
Bit 2 (Wert 4) = Layer mit Super-Bitmap.

bm2: Wenn bei den Flags das 2. Bit für ein Layer mit eigener Riesen-Bitmap gesetzt ist, wird hier die Adresse der zugehörigen Bitmap-Struktur eingetragen.

Funktion: Öffnet ein neues Layer über allen anderen Layers.

Beispiel: `lay&=CreateUpFrontLayer&(li&,bm&,lmix%,lmiy%,lmax%,lmay%,4,mb&)`

Ergebnis: Die Variable *Layer* erhält die Adresse der neu geschaffenen Layer-Struktur zugewiesen. Konnte das neue Layer nicht erzeugt werden, so wird Null zurückgegeben.

Format: `ScrollLayer(li,l,dx,dy)`

li (LNG): Zeiger auf eine LayerInfo-Struktur.
l (LNG): Zeiger auf Layer-Struktur.
dx (INT): Anzahl der Pixel, um die in X-Richtung gescrollt wird.
dy (INT): Anzahl der Pixel, um die in Y-Richtung gescrollt wird.

Funktion: Scrollt den Layer über eine Super-Bitmap

Beispiel: `CALL ScrollLayer&(li&,lay&,-1,Ø)`

Ergebnis: keines.

Anhang III

Die Programmdiskette

In diesem Buch sind über 80 kleinere und größere Programme vorgestellt worden. Das Abtippen von Programmen ist recht mühselig und vor allem auch nie fehlerfrei durchzuführen. Damit Sie sich diese Arbeit und unnötigen Frust ersparen, ist diesem Buch eine Diskette mit allen Programmen beigelegt.

Bevor Sie die Programme auf der Diskette starten können, sind jedoch einige kleinere Vorarbeiten notwendig. Zuerst die Grundregel Nummer eins für alle Original-Disketten: Legen Sie von der Diskette eine Kopie an und arbeiten Sie nur mit der Kopie der Diskette. Wenn Sie die Kopie angelegt haben, werden Sie feststellen, daß die Diskette fast voll ist. Damit Sie die Programme laufen lassen können, müssen Sie das Amiga-Basic darauf kopieren. Dazu ist jedoch nicht genügend Platz vorhanden.

Kopieren Sie deshalb die vollständige Schublade *Grundlagen* mit dem kompletten Inhalt auf eine zweite Diskette. Die Programme aus dieser Schublade bieten sich dafür an, da sie weder die Libraries aus der Schublade *bue* noch Grafiken aus der Schublade *Bild* einsetzen. Nun können Sie das Programm Amiga-Basic auf die Diskette kopieren.

Wenn Sie die Programme starten, kann es durchaus sein, daß Sie einige Male die Fehlermeldung (meistens vom Programm selber) erhalten, daß nicht genügend Speicher zur Verfügung steht. Um die Grafik-Möglichkeiten zu demonstrieren, wird bei einigen Programmen fast der gesamte freie Speicher eingesetzt. Dabei spielt es keine große Rolle, ob Ihr Amiga »nur« 512 Kbyte oder mehr Speicherkapazität hat. Die meisten Grafikausgaben können nur mit den unteren 512 Kbyte des Amiga arbeiten. Damit Sie den Programmen möglichst viel Speicher zur Verfügung stellen können, gehen Sie wie folgt vor. Ziehen Sie das Icon des Programmes, das Sie starten wollen, aus allen Fenstern heraus, und legen Sie es direkt auf die Workbench. Schließen Sie vor dem Programmstart alle Fenster. Sollte die Kapazität Ihres Rechners immer noch nicht ausreichen, liegt das wahrscheinlich daran, daß sich die RAM-Disk auf der Workbench befindet. Diese benötigt dann zu viel vom Speicher. Da bleibt Ihnen nur ein Ausweg, eine weitere Kopie der Workbench anzulegen. Auf dieser Kopie entfernen Sie aus der Startup-sequence die Anweisungen zum Anlegen einer RAM-Disk.

Das Programm *TopIcon* hat eine kleine Sperre eingebaut. Lesen Sie bitte im entsprechenden Kapitel nach, warum das so ist und wie Sie die Sperre beseitigen können. Im übrigen sind alle Programme mehrfach auf einem Amiga mit 512 Kbyte und auf einem Amiga 2000 mit 1 Mbyte getestet worden. Brechen Sie bitte die Programme nicht mit den üblichen Break-Befehlen ab. Da etliche Programme über die Libraries Systemspeicher reservieren, wäre sonst der reservierte Speicherbereich für die weiteren Programme verloren. Wenn das Programm keine anderen Anweisungen gibt, werden die Programme durch einen Druck auf die Leertaste abgebrochen. Sie müssen allerdings manchmal einen Augenblick auf das Programm-Ende warten, da eventuell ein Programmteil noch abgearbeitet wird. In den Beschreibungen zu den jeweiligen Programmen finden Sie genaue Informationen über den Programmablauf.

Beim Abspeichern von weiteren Grafiken achten Sie darauf, daß dazu keine Kapazität auf der Diskette ist. Sie müssen neue Grafiken also auf einer anderen Diskette ablegen.

Stichwortverzeichnis

Horst-Rainer Henning

Grafik mit AMIGA-BASIC

Horst-Rainer Henning hat über Amiga-Basic schon folgende Bücher veröffentlicht: »Programmieren mit Amiga-Basic«, »Programmier-Praxis Amiga-Basic«.

Dieses Buch ist speziell der Grafik-Programmierung unter Amiga-Basic gewidmet. Es zeigt Ihnen, wie man durch geschickten Einsatz der Systemroutinen alles das aus dem Amiga herausholt, was tatsächlich in ihm steckt. Dabei macht es keinen großen Unterschied, ob Sie sich zu den Einsteigern oder zu den fortgeschrittenen Programmierern zählen.

Im ersten Teil des Buches werden die Grafik-Befehle des Amiga-Basic behandelt. Dieser Abschnitt dient nicht nur als Hilfe für den Anfänger, sondern es werden auch Techniken vorgestellt, die oft vernachlässigt werden. So zum Beispiel »animierte« Grafik allein durch eine periodische Farbänderung.

Im zweiten Teil werden die Routinen des Betriebssystems zur Grafik-Programmierung miteinbezogen. Damit erweitern sich die Basic-Fähigkeiten um ein Vielfaches. Zusätzlich erreichen Sie eine enorme Steigerung der Verarbeitungsgeschwindigkeit, die vergessen läßt, daß es sich um ein Basicprogramm handelt. Programme, die das Double-Buffering benutzen, arbeiten dadurch schneller und natürlich eindrucksvoller, als normale Basic-Anwendungen es könnten. Linien-Grafiken und komplette Bildschirmausschnitte werden verschoben, vergrößert, verkleinert, gespiegelt oder verzerrt. Stellen Sie sich einmal vor, das Gesicht der Mona Lisa um eine Tonne gezeichnet, und das in Bruchteilen einer Sekunde!

Was Sie noch erwartet:
- Business-Grafik
- Bobs und Sprites, Hardware-Sprites
- Grafik-Routinen der »graphics.library«
- Super Bitmap
- Playfield Scrolling
- Dual Playfield
- ExtraHalfBrite (EHB)
- HAM – Hold And Modify
- Schnelles Speicher- und Ladeprogramm auch für HAM, EHB, SuperBM und Bildschirm-Ausschnitte
- Icon-Programmierung
- Fraktale Grafik

Die Begleitdiskette:

Sie enthält über 50 Beispielprogramme und erspart Ihnen das lästige und fehlerträchtige Abtippen der Listings.

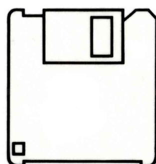
Hardware-Anforderungen:

Amiga 500, 1000, 2000 mit 512 Kbyte Arbeitsspeicher

Software-Anforderungen:

Amiga-Basic





ISBN 3-89090-669-9



DM 59,– sFr 54,30 öS 460,–